Generic programming in modern C++ for Image Processing

Michaël Roynard

Doctoral School n°130: École Doctorale Informatique, Télécommunications et Électronique (EDITE)

Doctoral Thesis

submitted to fulfill requirements for the degree of Doctor of

with the Doctoral Speciality of
"Software Engineering & Image Processing"

Generic programming in modern C++ for Image Processing

presented and defended to the public by

Michaël Roynard

on the Friday 4th November, 2022

*before distinguished members of the **Jury**:*

President	**Pr. Hugues Talbot**	CentraleSupélec / Université Paris Saclay
Reporter	**Pr. Benjamin Perret**	ESIEE / LIGM / Université Gustave Eiffel
Reporter	**Pr. Pascale Le Gall**	CentraleSupélec / Université Paris Saclay
Jury member	**Pr. Laurent Najman**	ESIEE / LIGM / Université Gustave Eiffel
Jury member	**Dr. Camille Kurtz**	LIPADE / SIP Lab / Université Paris Cité
Jury member	**Dr. Joël Falcou**	LRI / Université Paris Saclay
Director	**Pr. Thierry Géraud**	LRE / EPITA
Advisor	**Dr. Edwin Carlinet**	LRE / EPITA

Laboratoire de Recherche de l'EPITA (LRE)
14–16 rue Voltaire, 94270 Le Kremlin-Bicêtre, France

Acknowledgements

The past years doing my PhD program at the LRDE, EPITA was rich and intense in experience. I would like to thank all of those that have supported me through this journey, which is a PhD thesis.

I would like to sincerely thank Benjamin Perret and Pascale Le Gall who accepted to review my thesis. Also, I would like to thank Hugues Talbot for accepting to preside over my jury, as well as Laurent Najman, Camille Kurtz and Joël Falcou who agreed to be part of my jury.

I would like to address my sincere thanks to Thierry Géraud, my thesis director. You have recruited me and trusted me from the start. Even though the journey was bumpy and full of happenings, your advises and remarks were always on point, accurate and helpful until I defend my thesis in the best condition possible.

My sincere thanks also go to Edwin Carlinet, my thesis supervisor, which has given me invaluable help all along this journey. Thank you for pushing me forward when I needed it the most and for accompanying me to the end.

I wish especially to thank Claire Lecocq, for her support and advices that allowed to clarify the way to navigate to the end of this journey.

I would like to take the time to thank Daniela, who has accompanied me all through my thesis. You helped for my integration at the beginning and has always helped me, notably to get through all the administrative tasks at every step of the PhD.

Special tanks to Clément, our dearest Sysadmin. You were always on point, helpful and quick to make my computer work again! Thank you for your sysadmin jokes too! They really lift up the mood.

Also, I would like to thank all the members of the Image team, Jonathan, Joseph, Guillaume, Élodie, Nicolas and Olivier for their joyful mood and their help all along this thesis.

Of course, I would also like to thank the people of the Spot team (automata), especially Alexander, Sven, Jim, Adrien, Philipp, and Didier. It has been (and still is) a real pleasure to work with you.

I would like to warmly thank the PhD team, always there to share a worry, to give an advice, to drink a beer, all those essential things a PhD student needs to survive! A big thanks to Ludovic and Duy, who have set on their own path now. I wish you the best. A big thank you to Baptiste, Caroline, Minh, Charles, Zhou, Yizi, Antoine, Anissa and Florian for their support, especially the jokes and political debates during the breaks. Let us redo the world again around a beer next time!

Finally, I would like to thank my beloved Family, for their unfailing support, from the beginning to the end of this journey. You have always believed in me. Thank you for that. I would have not been able to finish without you.

Abstract

C++ is a multi-paradigm language that enables the initiated programmer to set up efficient image processing algorithms. This language strength comes from several aspects. C++ is high-level, which enables developing powerful abstractions and mixing different programming styles to ease the development. At the same time, C++ is low-level and can fully take advantage of the hardware to deliver the best performance. It is also very portable and highly compatible which allows algorithms to be called from high-level, fast-prototyping languages such as Python or Matlab. One of the most fundamental aspects where C++ really shines is generic programming. Generic programming makes it possible to develop and reuse bricks of software on objects (images) of different natures (types) without performance loss. Nevertheless, conciliating the aspects of genericity, efficiency, and simplicity is not trivial. Modern C++ (post-2011) has brought new features that made the language simpler and more powerful. In this thesis, we first explore one particular C++20 aspect: the concepts, in order to build a concrete taxonomy of image related types and algorithms. Second, we explore another addition to C++20, ranges (and views), and we apply this design to image processing algorithms and image types in order to solve issues such as how hard it is to customize/tweak image processing algorithms. Finally, we explore possibilities regarding how we can offer a bridge between static (compile-time) generic C++ code and dynamic (runtime) Python code. We offer our own hybrid solution and benchmark its performance as well as discuss what can be done in the future with JIT technologies. Considering those three axes, we will address the issue regarding the way to conciliate generic programming, efficiency and ease of use.

C++ est un langage de programmation multi-paradigme qui permet au développeur initié de mettre au point des algorithmes de traitement d'images. La force de langage se base sur plusieurs aspects. C++ est haut-niveau, cela signifie qu'il est possible de développer des abstractions puissantes mélangeant plusieurs styles de programmation pour faciliter le développement. En même temps, C++ reste bas-niveau et peut pleinement tirer partie du matériel pour fournir un maximum de performances. Il est aussi portable et très compatible ce qui lui permet de se brancher à d'autres langages de haut niveau pour le prototypage rapide tel que Python ou Matlab. Un des aspects les plus fondamentaux où le C++ brille, c'est la programmation générique. La programmation générique rend possible le développement et la réutilisation de briques logiciel comme des objets (images) de différentes natures (types) sans avoir de perte au niveau performance. Néanmoins, il n'est pas trivial de concilier les aspects de généricité, de performance et de simplicité d'utilisation. Le C++ moderne (post-2011) amène de nouvelles fonctionnalités qui le rendent plus simple et plus puissant. Dans cette thèse, nous explorons en premier un aspect particulier du C++20 : les concepts, dans le but de construire une taxonomie des types relatifs au traitement d'images. Deuxièmement, nous explorons une autre fonctionnalité ajoutée au C++20 : les ranges (et les vues). Nous appliquons ce design aux algorithmes de traitement d'images et aux types d'image, dans le but résoudre les problèmes liés, notamment, à la difficulté qu'il existe pour customiser les algorithmes de traitement d'image. Enfin, nous explorons les possibilités concernant la façon dont il est possible de construire un pont entre du code C++ générique statique (compile-time) et du code Python dynamique (runtime). Nous fournissons une solution hybride et nous mesurons ses performances. Nous discutons aussi les pistes qui peuvent être explorées dans le futur, notamment celles qui concernent les technologies JIT. Etant donné ces trois axes, nous voulons résoudre le problème concernant la conciliation des aspects de généricité, de performance et de simplicité d'utilisation.

Long summary

Introduction

Nowadays *Computer Vision* and *Image Processing (IP)* are omnipresent in the day-to-day life of the people. It is present each time we pass by a CCTV camera, each time we go to the hospital do an MRI, each time we drive our car and pass in front of a speed camera and each time we use our computer, smartphone or tablet. It cannot be avoided it anymore. The systems using this technology are sometimes simple and, sometimes, more complex. Also, the usage made of this technology has many purposes such as space observation, medical imaging, quality of life improvement, surveillance, control, autonomous system, etc. Henceforth, *Image Processing* has a wide range of research and despite having a mass of previous of work already contributed to, there are still a lot to explore.

Let us take the example of a modern smartphone application which provides facial recognition in order to recognize people whom are featuring inside a photo. To provide an accurate result, this application will have to do a lot of different processing through several steps. In addition, there are a lot of variables to handle. We can list (non exhaustively) the weather, the light exposition, the resolution, the orientation, the number of person, the localization of the person, the distinction between humans and objects/animals, etc. All of these elements needs to be carefully handled in order to finally recognize the person(s) inside the photo. What the application does not tell you is the complexity of the image processing pipeline behind the scene that, most of the time, cannot even be executed in its entirety on one's device (smartphone, tablet, ...). Indeed, image processing is costly in computing resources and would not meet the time requirement desired by the user if the entire pipeline was executed on the device. Furthermore, for the final part which is "recognize the person on the photo", the application needs to feed the pre-processed photo to a neural network trained beforehand through deep learning techniques in order to give an accurate response. There exists technologies capable of embedding neural network into mobile phone such as MobileNets [135], but it remains limited in terms of operational capabilities. It can detect a human being inside a photo but not give the answer about whom this human being is for instance. That is why, accurate neural network system usually are abstracted away in cloud technologies making them available only via Internet. When uploading his image, the user does not imagine the amount of technologies and computing power that will be used to find who appear on the photo.

We now understand that in order to build applications that interact with photos or videos nowadays, we need to be able to do accurate, fast and scalable image processing on a multitude of devices (smartphone, tablet, ...). In order to achieve this goal, image processing practitioners needs to have two kinds of tools at their disposal. The first is the prototyping environment, a toolbox which allow the practitioner to develop, test and improve its application logic. The second one is the production environment which deploys the viable version of the application that was developed by the practitioner. Both environment may not have the same needs. On one hand, the prototyping environment usually requires to have a fast feedback loop for testing, an availability of state-of-the-art algorithms and existing software. This way the practitioner can easily build on top of them and be fast enough so that he does not wait a long time to get the

results when testing many prototypes. On the other hand, the production environment must be stable, resilient, fast and scalable.

When looking at standards in the industry nowadays, we notice that the *Python* programming language is the main choice for prototyping. However, Python may not be suitable to push a viable prototype in production with minimal changes afterwards. We find it non-ideal that the practitioner cannot take advantages of many optimization opportunities, both in terms of algorithm efficiency and better hardware usage, when proceeding this way. It would be much more efficient to have basic low level building blocks that can be adapted to fit as mush use cases as possible. This way, the practitioner can easily build on top of them when designing its application. We distinguish two kinds of use cases. The first one is about the multiplicity of types or algorithms the practitioner is facing. The second one is about the diversity of hardware the practitioner may want to run his program. The goal is to have building blocks that can be intelligent enough to take advantage of many optimization opportunities, with regard to both input data types/algorithms and target hardware. Then the practitioner would have an important performance improvement, by default, without specifically tweaking his application. As such, the concept of genericity is introduced. It aims at providing a common ground about how an image should behave when passed to basic algorithms needed for complex applications. This way, in theory, one only needs to write the algorithm once for it to work with any given kind of image.

In the end, it is often known that there is a rule of three about genericity, efficiency and ease of use. The rule states that one can only have two of those items by sacrificing the third one. If one wants to be generic and efficient, then the naive solution will be very complex to use with lots of parameters. If one wants a solution to be generic and easy to use, then it will be not very efficient by default. Finally, If one wants a solution to be easy to use and efficient then it will not be very generic. To illustrate this rule, we can find examples among existing libraries. A notably generic and efficient library in C++ is Boost [169]: it is also notably known to be hard to use. Components such as Boost.Graph, Boost.Fusion or Boost.Spirit are hard to use. Also, a library which is generic and easy to use is the Json parser written by Niels Lohmann [172] it strives to handle every use case while remaining very easy to integrate and to use in user code (syntax really close to native Json in C++ code by providing DSL Domain Specific Language) [32] to parse C++ constructs into JSON). However, this has a cost and the parser is slower than Json parser optimized for speed such as simdjson [171] whose aim is to "parse gigabytes of JSON per second". Finally, there are plenty of example of user-friendly and efficient code which is not generic. We can cite Scikit-image [120] and OpenCV [29] that are easy to use and efficient (lot of handwritten SIMD/GPU code) but not generic due to the design choices.

In this thesis, we chose to work on an image processing library through continuing the work on Pylene [140]. But only working at library level would restrict the usability of our work and thus its impact. That is why we aim to reach prototyping users through providing a package that can be used in dynamic language such as Python without sacrificing efficiency. In particular, we aim to be usable in a Jupyter notebook. It is a very important goal for us to reach a usability able to permeate into the educational side which is a strength of Python. In this library, we demonstrate how to achieve genericity and efficiency while remaining easy to use all at the same time. In doing so, we are endeavoring to break through the rule previously cited. The scope of this library is limited to mathematical morphology [111, 83] and to the provision of very versatile image types. We leverage the modern C++ language and its many new features related to genericity and performance to break through this rule in the image processing area. Finally, we attempt, to bring low level tools and concepts from the static world to the high level and dynamic prototyping world for a better diffusion and ease of use, thanks to a bridge between those two worlds.

With this philosophy in mind, this manuscript aims at presenting our thesis work related to the C++ language applied to the Image Processing domain. It is organized as followed:

Generic Programming (genericity) We present a state-of-the-art overview about the notion of genericity. We explain its origin, how it has evolved (especially within the C++ language), what issues it is solving, what issues it is creating. We explain why image processing and genericity work well together. Finally, we tour around existing facilities that allows genericity (intrinsically restricted to compiled language) to exists in the dynamic world (with interpreted languages such as Python).

Taxonomy for Image Processing: Image types and Algorithms We present our first contribution in the image processing area which is a comprehensive work consisting in the taxonomy of different images families as well of different algorithms families. We explain, among others, the notion of concept and how it applies to the image processing domain. Also, we explain how to extract a concept from existing code, how to leverage it to make code more efficient and readable. Finally, we offer our take in the form of a collection of concepts related to image processing area.

Images Views We present our second contribution which is a generalization of the concept of View (from the C++ language, the work on ranges [143]) to images. This allows the creation of lightweight, cheap-to-copy images. It also enables a much simpler way to design image processing pipeline by chaining operations directly in the code in an intuitive way. Ranges are the cement of new designs to ease the use of image into algorithms which can further extend their generic behavior. Finally, we discuss the concept of lazy evaluation and the impacts of views on performance.

A bridge between the static world and the dynamic world We present our third contribution which is a way to grant access to the generic facilities of a compiled language (such as C++) to a dynamic language (such as Python) to ease the gap crossing between the prototyping phase and the production phase. Indeed, it is really not obvious to be able to conciliate generic code from C++ whose genericity is resolved at compilation-time (we call this the "static world"), and dynamic code from Python which rely on pre-compiled package binaries (we call this the "dynamic world"), to achieve an efficient communication between the dynamic code and the library. We also cannot ask of the user to provide and use a compiler each time he wants to use our library from Python. We discuss what are the existing solutions that can be considered as well as their pros. and cons. Finally, we discuss how we designed a hybrid solution to make the bridge between the static world and the dynamic world.

Generic programming (genericity)

In natural language we say that something is generic when it can fit several purposes at once while being decently efficient. For instance, a computer is a generic tool that allows one to write documents, access emails, browse Internet, play video games, watch movies, read e-books etc. In programming, we will say that a tool is generic when it can fit several purposes. For instance, the gcc compiler can compile several programming languages (C, C++, Objective-C, Objective-C++, Fortran, Ada, D, Go, and BRIG (HSAIL)) as well as target several architectures (IA-32 (x86), x86–64, ARM, SPARC, etc.). Henceforth, we can say that gcc is a generic compiler. At this point it is important to note that even though a tool is deemed generic, there is a scope on what the tool can do and what the tool cannot do. A compiler despite supporting many languages and architectures, will not be able to make a phone call or a coffee. As such it is important to note that genericity is an aspect that qualifies something. We study the generic aspects related to libraries and programming languages.

This thesis voluntary leaves out the generic aspect related to the target architecture. Indeed, being able to write and/or generate code that is able to run on a large array of different hardware

architecture is a field of research on its own and is not the main focus of this thesis. It is also known as *heterogeneous computing*. Instead, we will focus on the aspects related to genericity at a library level and at a programming language level.

Genericity within libraries It is described by the cardinality of how many use-cases it can handle. Libraries always provides their own data structures, to represent and to give a meaning to the data the user wants to process, as well as algorithms to process those data and provide different type of results. A library will be then labeled as *generic* [15] when (i) its data structures allow the user to express himself fully with no limitation and when (ii) its algorithm bank is large enough to do anything the user would want to do with its data. In reality such a library does not exist and there are always limitations. Studying those limitations and what reasons motivate them is the key to understand how to surpass them in the future, by developing new hardware and/or software support for new features enabling more genericity.

Genericity within programming languages It is described by the ability of the language to execute the same code over a large amount of data structures [31], be they native (char, int, ...) or user defined. It is nowadays primordial for a programming language to be able to do so. Indeed, in a world where Information Technologies are everywhere, the amount of code written by software developers is staggering. And with it so is the amount of bugs and security vulnerabilities. Being able to natively have a programming language that enables to do *more* by writing *less* mathematically results in a reduced development and maintenance cost. Programming languages offers many ways to achieve genericity which is dependent of the language intrinsic specificities: compiled or interpreted, native or emulated, etc.

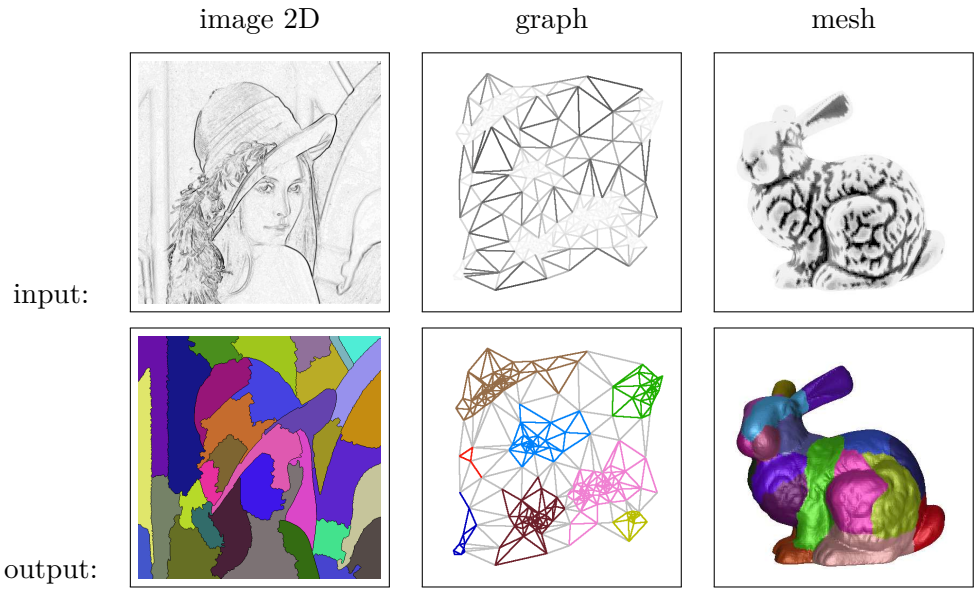
Before delving into the specifics of what genericity implies for libraries and programming languages, let us introduce some vocabulary for the sake of comprehension. First is the notion of *type*. A *type* (or *data type*) is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data. Most programming language support basic data types (also called primitive types) such as integer numbers, floating point numbers, boolean and characters (ASCII, Unicode, etc.). This data attribute defines the operations that can be performed on the data, the meaning of the data and the size of the data in memory (the data can then be stored on the heap, stack, etc.). A data type provides a set of values from which an expression (i.e. variable, function, etc.) may take those values. Among programming language, we can distinguish those who are dynamically types and those that are statically typed. Statically typed languages are those whose variables are declared holding a specific type. This variable cannot hold data from another type in the scope it is declared. Statically typed programming languages are Ada, C, C++, Java, Rust, Go, Scala. Dynamically types languages are those whose variables can be reassigned with a value of different type from the one it was initially declared to hold. The variable type is then dynamically changed to fit the new value it is holding. Dynamically typed programming languages are PHP, Python, JavaScript, Perl.

The consequence of being able to tell which type a variable is holding at all time (statically-typed language) is two-fold. For the developer, it is easier to reason about code and to spot bugs. For the compiler, it is possible to generate optimized binary code specific to this data type (vectorization, etc.). The consequence of being able to morph the type a variable can hold at runtime is mainly to serve prototyping purpose. When tweaking a Jupyter notebook, it is much appreciated not to be limited to a single type for each variable to be able to iterate on the prototype much faster.

In image processing, an image Im is defined on a domain $\mathcal{D}$ (which contains points) by the relation $\forall x \in \mathcal{D}, y = Im(x)$ where y is the value of the image Im for the point x. This definition always translates into a complex data structure when transposed into a programming language. This data structure must be aware of the data buffer containing the image data as well as information about the size and dimensions of the image. Furthermore, to add to the difficulty,

the information needed to define precisely the data structure is not always known when writing the source code. Indeed, a very simple use-case consists in reading an image from a file to load it in memory. The file can contain an image of varying data type and the program should still work properly. There are multiple approach to solve this issue, and we are addressing them.

Projecting the notion of genericity to Image Processing, we can deduce that we need two important aspects in order to be generic. First, we need to decorrelate the data structures and its topology and underlying data from the algorithms. Indeed, we want our algorithms to support as much data structures as possible. Second, many algorithms share the same computational shape and can be factorized together.



The same code run on all these inputs.

Figure 1: Watershed algorithm applied to three different image types [118].

Genericity can have two different meanings depending on the people you ask. For instance, some will argue that genericity is a high level aspect and will qualify a tool by "generic enough" when it handles all of his use-cases. Others will argue that genericity is a low level aspect that relates to the machine (code) building the tools, "generic enough" to make a lot different tools. Neither is wrong. However, for the sake of comprehension we will use different words for each of these cases. A tool generic enough to handle a lot of use-case will be called *versatile*. Finally, for a tool whose aim is to be able to build a lot of different tools (i.e. providing a programming framework able to handle code of any use-case) we will use *generic*. In this thesis, genericity will be about code, meaning the programming framework able to handle any use-case. The fig. 1 [92] illustrates the result of the same generic watershed implementation applied on an image 2D, a graph as well as a mesh.

We present the origin of generic programming, which goes as far as 1988, year and how it has evolved to be integrated in the Ada programming language and then the C++ programming language. Afterwards, it has evolved even further with the notion of *concept* which completes the toolbox required to be able to fully make use of generic programming without resorting to obscure tweaks and tools.

We explore the possibilities of achieving the notion of genericity from within a library. Indeed, there are three techniques enabling the user to write a high level algorithm once that can run on every type. They are the *code duplication* approach, the *generalization* approach and the *inclusion and parametric polymorphism* approach [72].

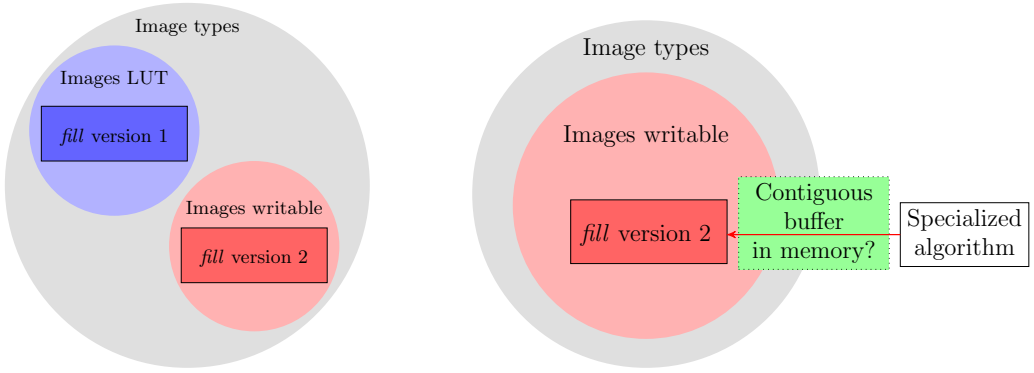
Finally, we present the inherent limitation of C++ templates, which is that they remain in the static world (compile time). Genericity (in the sense C++ template) does not exist in the final shipped binary to the user. The final user, in its dynamic world (runtime) cannot use a generic (C++ code) tool. We discuss the different approaches possible to bridge this gap between the static (compile-time) and dynamic (runtime) world.

We will then make extensive use of Genericity to present the first contribution of this thesis: a taxonomy of concepts related to Image processing.

Taxonomy for Image Processing: Image types and Algorithms

In this thesis, we have pursued research into how to apply all those new generic facilities from the C++ language into the Image processing area. This allows us to test them in a practical way on our predilection area while remembering our past work, both success and failures in this matter. However, birthing concepts from code is something that is done in an emerging way. Henceforth, the first work will be to do an inventory of all existing image algorithms as well as an inventory of all image processing algorithms (both basic and more complex) we can think of. This way, we will notice behavior patterns emerging from similar image types or similar algorithms. We will then be able to extract behavioral patterns from this inventory in order to produce a full taxonomy in the form of a framework of concepts related to image processing.

We present that concepts are not designed after data structures but after algorithms. Indeed, a concept consists in extracting a consistent behavioral pattern from a piece of code (algorithm) and name it to give him a meaning. Through a simple but concrete example, we present in a didactic way how to extract concepts from an image processing algorithm (gamma correction).



(a) Different versions of *fill* algorithm

(b) Specialization existing within a version

Figure 2: Set of algorithm versions (a) and its specialization existing within a version (b).

We then explain how, in theory, image types are related to each other. We present the set of different image types and how algorithms exist in those sets, which introduce the notion of *version* of an algorithm. An algorithm will have different *versions* for each image types set it supports. We distinguish it (in fig. 2) from an algorithm *specialization*, the latter being the ability to use an opportunity (related to a property) to make an optimization and increase performances.

We then describe the notion how algorithm canvas which is the result flowing from the taxonomy of image processing algorithms. Indeed, there are three main algorithm families: the pixel-wise algorithms (binary threshold), the local algorithms (dilation) and the global algorithms (Chamfer distance transform). We focus primarily on local algorithms and how they can all be written through the same canvas of code. Indeed, for instance, the only difference between a dilation and an erosion is the operator (max vs. min). We then discuss ways to exploit these canvas to possibly solve heterogeneous computing issues.

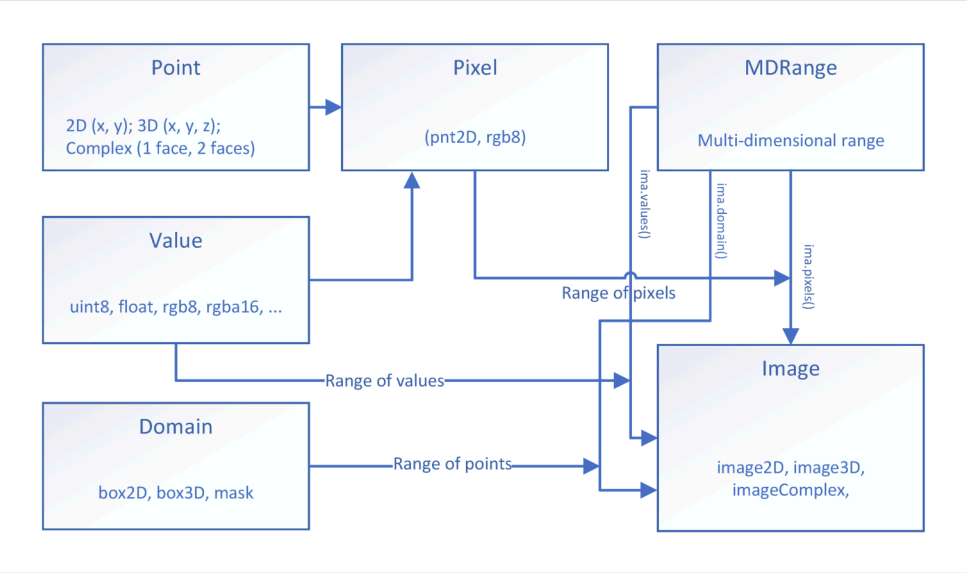


Figure 3: Image concept.

Finally, we introduce our first main contribution: a complete taxonomy related to the image processing area. We first introduce fundamental concepts such as *point*, *pixel*, *domain* and *image* (illustrated in fig. 3). We then motivate and introduce advanced concepts related to images and the different way to access data (forward, backward traversing, indexing, direct access to underlying buffer, ...). In the end, we introduce the concepts related to orbiting notions such as *structuring element*, *neighborhood* and *extension* (border management) which are necessary to be able to work with local algorithms.

We will then make use of the presented concepts to introduce the second main contribution of this thesis: the *image views*.

Image views

This concept of views is not new [24] and naturally appeared in Image processing with Milena [96, 85] under the name of *morpher* [80, 95]. It was always useful to be able to project an image through a prism that could extract specific information about it without the need to copy the underlying data buffer. In modern days, the language C++ (20) also introduces this mechanism with the ranges [184] facilities for *non-owning collections*. It is named *views* and allows the user to access the content of a container (vector, map) through a prism. In Pylene, we decided to align the naming system after what was decided in C++20 in order not to confuse the user. This way, a `transform` view in image processing will do the same thing on an image that the transform view in the standard range library does on a container. *Views* feature the following properties: *cheap to copy*, *non-owner* (does not *own* any data buffer), *lazy evaluation* (accessing the value of a pixel may require computations) and *composition*. When chained, the compiler builds a *tree of expressions* (or *expression template* as used in many scientific computing libraries such as Eigen [84]), thus it knows at compile-time the type of the composition and ensures a 0-overhead at evaluation.

In image processing an algorithm is naively written by taking one or several inputs' data (among which is the input image(s)), by performing work on this input data and then by returning the resulting data (or an error). Let us take for example the alpha-blending example which can be implemented in naive C++ code as followed:

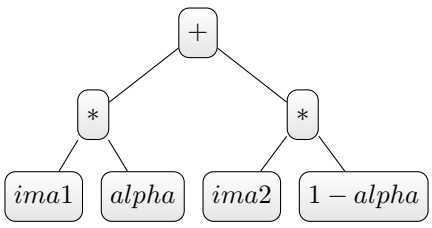
```
void blend_inplace(const uint8_t* ima1, uint8_t* ima2, float alpha,
int width, int height, int stride1, int stride2) {
  for (int y = 0; y < height; ++y) {
```

```
      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);
    }
  }
```

This code has several flaws. It makes strong hypothesis about the input images: its data buffer contiguity and its shape (2D). Let us suppose that our user now wants to restrict the algorithm to a specific region inside the image. The maintainer would have then to provide an overload of the algorithm with one additional input argument corresponding to the region of interest. Let us suppose that the user now wants to support manipulate 3D images. The maintainer would now have to provide two additional overloads with an additional stride argument (one for the base algorithm, one for the region of interest-restricted algorithm). Let us now suppose that the user only wants to manipulate the red color channel. Now the maintainer must support and add additional overloads of the algorithm for each channel and/or color type. The complexity increases manyfold for each kind of customization points the maintainer wants to offer to the user. Of course, it is possible to prevent code duplication through clever usage of computer engineering techniques (code factorization etc.) but the complexity would still leak through the API anyway. That is way the other solution is to make the user able to perform those restriction upstream from the algorithm transparently so that the downstream algorithm is easy to write, understand and maintain. In order to achieve this, we need to raise the abstraction level around images by one layer so that we can work at the image level. The alpha-blending algorithm would then be written as shown in fig. 4.

Figure 4: Alpha-blending algorithm written at image level.

This way to express an algorithm is achieved by introducing *views* to image processing. An image now is a view and can be restricted/projected/manipulated however the user need before feeding it to an algorithm. Even the whole alpha blending algorithm can be rewritten in terms of views entirely, as shown in fig. 5.




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

Figure 5: Alpha-blending, generic implementation with *views*, expression tree.

Being able to perform powerful manipulation on images before feeding them to algorithms completely nullify the initial problem of having several overloads of the same algorithm while maintaining and documenting all the associated optional arguments. Indeed, in order to perform the alpha-blending transformation on the base input image, all that the user must do is:

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

If the user wants to restrict the region to be blended, or the color channel to work on, he just has to write the following modification:

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

The restriction is done upstream from the algorithm and propagated downstream without increasing the code complexity. This way, view greatly increase what the user can do by writing less code.

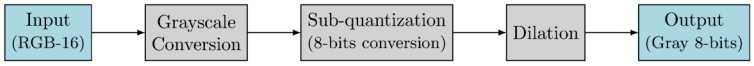


Figure 6: Example of a simple image processing pipeline.

We see that *views are composable*. One of the most important feature in a pipeline design (generally, in software engineering) is *object composition*. It enables composing simple blocks into complex ones. Those complex blocks can then be managed as if they were still simple blocks. In fig. 6, we have 3 simple image processing operators *Image* → *Image* (the grayscale conversion, the sub-quantization and the dilation). As shown in fig. 7, algorithm composition would consider these 3 simple operators as a single complex operator *Image* → *Image* that could then be used in another even more complex processing pipeline. Just like algorithms, image views are composable, e.g. a view of the view of an image is still an image. In fig. 7, we compose the input image with a grayscale transform view and a sub-quantization view that then feeds the dilation algorithm.

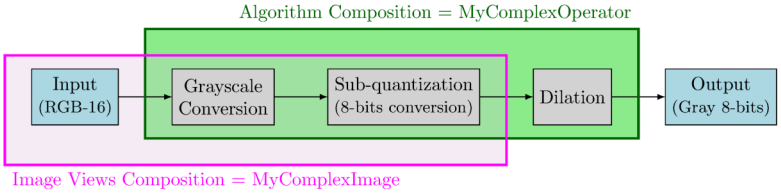


Figure 7: Example of a simple image processing pipeline illustrating the difference between the composition of algorithms and image views.

Also, *views improve usability*. The code to compose images in fig. 7 is almost as simple as:

```
auto input = imread(...);
auto A = transform(input, [](rgb16 x) -> float {
  return (x.r + x.g + x.b) / 3.f; }; );
auto MyComplexImage = transform(A, [](float x)
  -> uint8_t { return (x / 256 + .5f); }; );
```

People familiar with functional programming may notice similarities with these languages where *transform* (*map*) and *filter* are sequence operators. Views use the functional paradigm and are created by functions that take a function as argument: the operator or the predicate to apply for each pixel; we do not iterate by hand on the image pixels.

Furthermore, *views improve re-usability*. The code snippets above are simple but not very re-usable. However, following the functional programming paradigm, it is quite easy to define new views, because some image adaptors can be considered as *high-order functions* for which we can bind some parameters, as one would do with the curry technique [17]. In fig. 8, we show how the primitive *transform* can be used to create view operators summing two images, performing the grayscale conversion and performing a sub-quantization. This basic views, which can be reused afterwards, are then chained together to create a complex image. (These functions could have been written in a more generic way for more re-usability, but this is not the purpose here.)

```
auto operator+(Image A, Image B) {
  return transform(A, B, std::plus<>());
}
auto togray = [](Image A) { return transform(A, [](auto x)
  { return (x.r + x.g + x.b) / 3.f; };)
};
auto subquantize16to8b = [](Image A) { return transform(A,
  [](float x) { return uint8_t(x / 256 +.5f); });
};

auto input = imread(...);
auto MyComplexImage = subquantize16to8b(togray(A));
```

Figure 8: Using high-order primitive views to create custom view operators.

In addition, *views are lazy computed*. Because the operation is recorded within the image view, this new image type allows fundamental image types to be mixed with algorithms. In fig. 8, the creation of views does not involve any computation in itself but rather delays the computation until the expression `v(p)` is invoked. Because views can be composed, the evaluation can be delayed quite far. Image adaptors are *template expressions* [20, 40] as they record the *expression* used to generate the image as a template parameter. A view actually represents an expression tree (fig. 5).

Also, *views are efficient*. With a classical design, each operation of the pipeline is implemented on "its own". Each operation requires memory to be allocated for the output image and also, each operation requires that the image is fully traversed. This design is simple, flexible, composable, but is not memory efficient nor computation efficient. With the lazy evaluation approach, the image is traversed only once (when the dilation is applied) which has two benefits. First, there are no intermediate images which is very memory efficient. Second, traversing the image is faster thanks to a better memory cache usage, and performs an optimal selective traversal. Indeed, in our example (fig. 6), processing a RGB16 pixel from the dilation algorithm directly converts it in grayscale, then sub-quantize it to 8-bits, and finally makes it available for the dilation algorithm. It acts *as if* we were writing an optimal operator that would combine all these operations. This approach is somewhat related to the kernel-fusing operations available in some HPC specifications [150] but views-fusion is optimized by the C++ compiler only [139]. The selective aspect intervenes when a region of interest is selected at one point in the processing pipeline. Indeed, the entirety of the pipeline is then executed only on the region of interest, even if this selection happens only at the very end of the processing pipeline.

Finally, *views improve the productivity*. All point-wise image processing algorithms can (and should) be rewritten intuitively by using a one-liner view. The *transform* views is the key enabling that point. This implies that there exist a new abstraction level available to the practitioner when prototyping their algorithm. The time spent implementing features is reduced, thus the feedback-loop time is reduced too. This naturally brings productivity gain to the practitioner.

A bridge between the static world and the dynamic world

In the programming world, there are three main families of programming language [38]. There are (i) *compiled* programming languages, such as C, C++, Rust or Go, (ii) *interpreted* programming languages, such as Python, PHP, Lisp or Javascript, and (iii) hybrid programming languages, such as Java or C#. The latter have a fast compilation pass that compiles the source code into an intermediate bytecode. Then, this bytecode is interpreted via an interpreter on the host (runner) machine.

We design many solutions to solve several kinds of issues related to the bridge between the static and the dynamic world. We present our hybrid solution that is able to make our C++ templated library (static) available from Python (dynamic). We discuss several ways of achieving

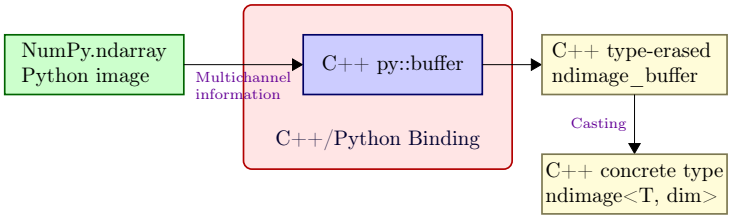


Figure 9: Bridge from Python to C++ via Pybind11 and a type-erased C++ class.

this feat, their pros. and their cons. Also, we introduce a new abstraction layer, the value-set, which is a standard way to manipulate an underlying image value usable when implementing image processing algorithms. This new abstraction layer enables the user to inject code from Python-side into already-complied C++ routines. However, layering one abstraction layer after another, or even calling Python code does come with performance cost. This is why we have run a benchmark to outline the cost of our solutions. This benchmark compares the four version of our stretch algorithm whose implementation and usage are detailed in the manuscript. The result is shown in table 1.

Dispatch type	Compute Time	ΔCompute Time
Native value-set with native C++ value-type (baseline)	0.0093s	0
Value-set with virtual dispatch with native C++ value-type	0.1213s	×13
Value-set with virtual dispatch with C++ type-erased values	1.0738s	×115
Injected Python value-set with native C++ value-type	21.5444s	×2316

Table 1: Benchmarks of all our version of the stretch algorithm.

This benchmark shows that each time an abstraction layer is added on top of the baseline, the user must expect a 10× slowness factor in his code performance. Also, calling Python code is immensely slower (2300× !) than the baseline. This renews the interest there is to recompile the templated C++ library with an additional known type than injecting it from Python for code taking long time to execute. Being able to inject Python code ease prototyping and increase the speed at which the user can write his code. However, the benchmark shows that this is not a viable solution once the prototype needs to scale to a production environment.

Continuation: JIT-based solutions, pros. and cons. Our hybrid solution certainly has advantages, but the huge disadvantage is the slowness of injecting our own types from the Python side. There exists another solution that this thesis did not have the opportunity to study in-depth. This solution is based on a known technology: the Just-In-Time (JIT) compilation which has been previously illustrated in fig. 10 (and which itself is based on the notion of generative programming [30]). Library such as AsmJit [91] are able to emit machine code directly by making call in C++ code. Indeed, it is a technology already used by interpreted languages such as Java or PHP to generate on-the-fly native and optimized machine code for the section of the source code that is considered “hot” by the interpreter. A source code is “hot” when it is executed a lot: the end-user would gain paying the compilation time once to have this code executed faster several times later on. When applying this strategy to our problematic, it would mean that the user must be able to compile native machine code from the templated generic C++ code by injected the requested type when it is used. Such an operation shift heavily the burden on the user, and it is well-known that compiling C++ code is notably *complicated* and *slow*. In addition, the library needs to be able to auto-generate Python binding once the C++ code is compiled, and to handle *NumPy.ndarray* types in the interface. There are several solutions to reach this point.

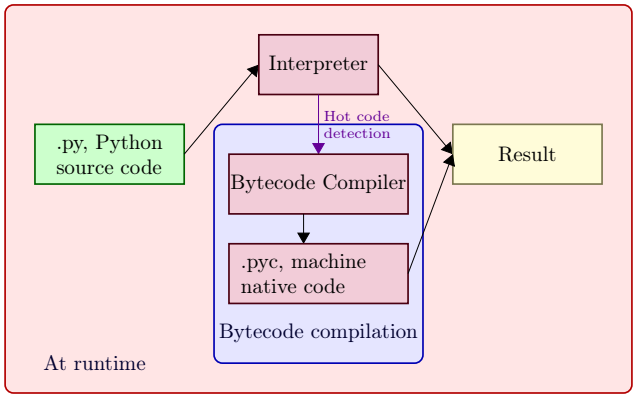


Figure 10: Interpreted languages: runtime

The first solution is to basically use system call to the compilers to actually *compile* C++ code once the templated types are known and explicitly instantiated in the source code. This solution requires careful code-generation design and that the user actually possess a compiler on his computer. Furthermore, the user must resolve all the library dependencies, such as *freeimage* for IO etc. This solution was engineered in the VCSN library [108]. Indeed, each time the user declared a new automaton in his Jupyter notebook, corresponding source code is compiled in the background and then cached. It is a very perilous solution to implement when the final execution environment (OS, installed software) is not well-known in advance. Nowadays, the issue may be lesser, however, it still requires to maintain both the library and the container solution to use it (Docker).

The second solution is to use Cython [88]. It is a transpiling infrastructure which transform a Python source code directly into C-language source code so that it can be compiled by a standard C compiler just by linking against the Python/C API. This remove the burden of writing the careful code-generation routine, system-calls to the C++ compiler and removes the need to resolve all the dependencies. This infrastructure takes care of everything for the user. Cython even support C++ template code [176] which is mandatory for our use-case.

The third solution consists in relying on recent projects that are all relying on the LLVM infrastructure. We can notably note AutoWIG [141], Cppyy [130], Xeus-cling [173] and Pythran [123]. AutoWIG has in-house code based on LLVM/Clang to parse C++ code in order to generate and compile a Swig Python binding using the Mako templating engine. AutoWIG, coupled with Cython would permit the user to, for instance, generate C code related to a custom Python structure. Then a simple call to AutoWIG will parse the C code and inject it into the C++ library to generate the appropriate bindings for the user. As for Cppyy, it is based on LLVM/Cling, a C++ interpreter, and can directly interpret C++ code from a Python string. This enables injecting custom types easily, be they in Python code (transpiled with Cython) or C++ code (directly interpreted by Cling). Afterwards, the infrastructure generates the appropriate binding from the templated C++ library for the injected type. Xeus-cling is a ready-to-use Jupyter kernel and allow the usage of C++ code directly from within a notebook. This completely bypass the need of a Python binding in the first place and allow the user to use the library from within the notebook as if he was using a Python library. Finally, Pythran is an ahead of time compiler for a subset of the Python language, with a focus on scientific computing. It takes a Python modle annotated with a few interface descriptions and turns it into a native Python module with the same interface, but hopefully faster. Pythran takes advantages of multicore and SIMD instructions to turns its subset of the Python language into heavily templated C++ code instantiated for particular optimized types. All those infrastructures, however, come with a hefty cost in terms of binary size. Indeed, a C++ compiler is not small and embarking it alongside the image processing library can easily impact greatly the final binary. Without the

LLVM infrastructure the binary may weight around 3MB. With the LLVM infrastructure, the binary weight at the bare minimum 50MB. Also, these solutions may not be immediately faster. Indeed, when prototyping back and forth with a variety of types, the user may not be eager to wait for long compilations times each time he is testing with a new iteration of his work. Despite those facts, those solutions offers great avenue of research for the future and the author is eager to explore them.

Conclusion

The work presented in this thesis by the author followed a very clear narrative arc. The emphasis was first shown on presenting what is the notion of generic programming (genericity), its story and how anyone can relate in his day-to-day life, especially when applied to image processing. Genericity is a 4-decades years old notion that has evolved and found usage in very modern areas of our society. Indeed, image processing is widely used to build modern applications used all around the world. However, it was demonstrated how difficult it can be to implement solutions relying on genericity. Indeed, there is a rule of three tying genericity, performance and ease of use stated in introduction. In this thesis, we try to demonstrate how to break this rule in three steps.

The first step, illustrated in *Taxonomy for Image Processing*, was to make an inventory of image types and families as well as different image processing algorithms. The aim was to produce a comprehensive taxonomy of types (pixel, image, structuring elements, ...) and algorithms related to image processing in order to be able to write concepts (in the sense of C++ concepts). This first step delimits the perimeter of what the author means by *genericity*. From this starting point, it becomes easier to write image processing algorithms, just by relying on those concepts. Furthermore, different concepts exist to enable algorithm implementers to leverage properties (structuring elements' decomposability, image's buffer contiguous, ...) in order to achieve maximum performance.

At this point, we are still reasoning at a low level (pixel) which generates the need to design an abstraction layer in order to enable fast prototyping for simple operations while guarantying very small memory footprint and near-zero performance impact. For this reason, we expand the concept of *views* from the C++ standard (2020) to images and design what the notion of *image view*. We also make the design choice to have cheap-to-copy image (shared data buffer) by default in order to merge concrete image and views from the user point of view. The lazy-evaluation, that systematically happens when using views allows performance gain when clipping larges images. In the case where the whole image is processed, we were able to still retain very satisfactory performance that remain stable. Also, we show through concrete use-case, such as pixel-wise algorithm and border management how the usage of views simplify greatly how to write more complex image processing algorithms that are efficient by default. We finally show the limitations of this approach, with a particular focus on the speed of traversing an image, which a mandatory use-case we must get right.

Finally, this thesis focused its attention on how it is possible to distribute this software to the image processing community which is mainly working with Python. The last work concentrates its efforts on finding the best way to design a static (compile-time, templated C++) — dynamic (runtime, Python notebook) bridge to bring those notions (concepts and views) to the practitioner, efficiently. This last work explores this dilemma and offers to address it with one hybrid solution whose design is explained in-depth. This hybrid solution rely on a type-erased type which offers compatibility with a *NumPy.ndarray* [161]. It is then able to cast itself inside an $n \times n$ dispatcher (dimension and underlying type) into an optimized concrete templated C++ type. This solution also explain how to write very simply the glue code enabling already-existing algorithms (in C++) to be exposed in Python thanks to a dispatch mechanic heavily inspired from the C++ standard (`std::visit`, `std::variant`). The aim of this solution was to regroup at a single place in the code all the supported types into the dispatchers for maintenance purpose as well as

demanding minimal work from algorithm implementer to expose their algorithms, all this while keeping the native performance. Indeed, no superfluous copy is performed thanks to *pybind11*'s *type-caster* facility, and one cast is done from the type-erased type to the native one. All the work that is done in the algorithm is performed on native optimized type. Finally, this solution offers a way to inject custom Python types into the library for prototyping purpose, thanks to a new abstraction layer, at the cost of heavy performance loss. The downside of this solution is obviously the code bloat with the resulting binary size exploding exponentially with the number of supported types multiplied by the number of algorithms multiplied by the number of additional supported data (structuring elements, label map, etc.)

We conclude this thesis by offering new avenue of research around the Just-In-Time (JIT) compilation area to further improve the bridge between the static and the dynamic world. The author thinks that this avenue is worth exploring, especially with the already promising existing tools (Xeus-cling, Cppyy, Cython, AutoWIG, Pythran) in order to solve the code bloat issue. Indeed, we would only compile what the user need, but the entry price may be to statically link a C++ interpreter (LLVM/cling?) into the binary which in itself would greatly bloat it. It may be possible, however, to rely on the user's system-wide infrastructure so that the maintenance does not distribute a whole C++ interpreter/compiler alongside his image processing library binary. This is still a new area of research and the author would very much want to delve into it to study what is possible to achieve as of today with those tools for the image processing community.

Résumé long

Introduction

De nos jours, la *Vision par ordinateur* et le *traitement d'images (PI)* sont omniprésents dans la vie quotidienne des gens. Ils sont présents à chaque fois que nous passons devant une caméra de vidéosurveillance, à chaque fois que nous allons à l'hôpital faire une IRM, à chaque fois que nous conduisons notre voiture et passons devant un radar et à chaque fois que nous utilisons notre ordinateur, smartphone ou tablette. Ils sont devenus incontournables. Les systèmes utilisant cette technologie sont souvent simples et parfois plus complexes. Cette technologie s'utilise dans des domaines très variés, tel que l'observation spatiale, l'imagerie médicale, l'amélioration de la qualité de vie, la surveillance, le contrôle, les systèmes autonomes, etc. Désormais, le *traitement d'images* dispose d'un large éventail de sujets de recherche et malgré une masse de travaux antérieurs déjà contribués très importante, il reste encore beaucoup à explorer.

Prenons l'exemple d'une application smartphone moderne qui propose une reconnaissance faciale afin de reconnaître les personnes qui figurent dans d'une photo. Pour fournir un résultat précis, cette application devra faire beaucoup de traitements différents en plusieurs étapes. De plus, il y a beaucoup de variables à gérer. Nous pouvons énumérer (non exhaustivement) la météo, l'exposition lumineuse, la résolution, l'orientation, le nombre de personnes, la localisation de la personne, la distinction entre les humains et les objets/animaux, etc. Tous ces éléments doivent être soigneusement gérés afin de reconnaître enfin la ou les personne(s) figurant dans la photo. Ce que l'application ne vous dit pas, c'est la complexité de la pipeline de traitement d'images en coulisse qui, la plupart du temps, ne peut même pas être exécutée dans son intégralité sur son appareil (smartphone, tablette, ...). En effet, le traitement d'images est coûteux en ressources informatiques et ne répondrait pas à la contrainte de temps demandée par l'utilisateur si l'intégralité de la pipeline était exécutée sur l'appareil. Par ailleurs, pour la dernière partie qui est de « reconnaître la personne sur la photo », l'application doit donner la photo pré-traitée à un réseau de neurones formé au préalable par des techniques d'apprentissage profond afin de donner une réponse précise. Il existe des technologies capables d'intégrer un réseau de neurones dans un téléphone mobile, telles que MobileNets [135], mais il reste limité en termes de capacités opérationnelles. Il peut détecter un être humain à l'intérieur d'une photo et même identifier la personne, mais par exemple, ne peut pas avoir la même capacité de distinction entre un grand nombre des personnes que les réseaux utilisés par les réseaux sociaux. C'est pourquoi les systèmes de réseau de neurones précis sont généralement hébergés dans le cloud, ce qui ne les rend disponibles que via Internet. Lors du téléchargement de son image, l'utilisateur n'imagine pas la quantité de technologies et de puissance de calcul qui va être utilisée pour reconnaître la personne apparaissant sur la photo.

Nous comprenons maintenant que pour créer, aujourd'hui, des applications qui interagissent avec des photos ou des vidéos, nous devons pouvoir effectuer un traitement d'images précis, rapide et évolutif sur une multitude d'appareils (smartphone, tablette, ...). Afin d'atteindre cet objectif, les traiteurs d'image doivent disposer de deux types d'outils. Le premier est l'environnement de prototypage, une boîte à outils qui permet au traiteur d'image de développer, tester et améliorer sa logique applicative. Le second est l'environnement de production qui déploie la version viable de l'application qui a été développée par le traiteur d'image. Les deux environnements peuvent ne

pas avoir les mêmes besoins. D'une part, l'environnement de prototypage nécessite généralement de disposer d'une boucle de rétroaction rapide pour les tests et d'une disponibilité des algorithmes des logiciels existants à la pointe des connaissances actuelles. De cette façon, le traiteur d'image peut facilement construire ses nombreux prototypes par-dessus ces briques de base (algorithmes existants) pour obtenir des résultats rapidement dans le but de pouvoir itérer de manière agile. D'autre part, l'environnement de production doit, quant à lui, être stable, résilient, rapide et évolutif.

Lorsque l'on regarde les standards de l'industrie aujourd'hui, nous remarquons que le langage de programmation *Python* est le principal choix pour le prototypage. Cependant, Python peut ne pas convenir pour pouvoir déployer un prototype viable en production avec un minimum changements par la suite. Nous trouvons qu'il n'est pas idéal que le traiteur d'image ne puisse pas profiter de nombreuses opportunités d'optimisations, à la fois en termes d'efficacité algorithmique et à la fois au niveau d'une meilleure utilisation du matériel. Il serait beaucoup plus efficace d'avoir des briques de construction de base (bas niveau) qui pourraient être adaptés pour convenir à autant de cas d'utilisation que possible. De cette façon, le traiteur d'image pourrait facilement s'appuyer sur ces briques lors de la conception de son application. Nous distinguons deux types de cas d'utilisation. Le premier concerne la multiplicité des types ou des algorithmes auxquels le traiteur d'image est confronté. Le deuxième relève de la diversité du matériel sur lequel il peut vouloir exécuter son programme. L'objectif est de concevoir intelligemment les briques de base qui permettent de tirer parti des nombreuses opportunités d'optimisation, tant en ce qui concerne les types de données et d'algorithmes en entrée, que le matériel cible. Ensuite, le traiteur d'image verrait une importante amélioration des performances, par défaut, sans devoir ajuster spécifiquement son application. C'est ainsi que le concept de généricité est introduit. Il vise à fournir un terrain d'entente sur la façon dont une image doit se comporter lorsqu'elle est transmise à des algorithmes de base nécessaires pour des applications complexes. De cette façon, en théorie, il suffit d'écrire l'algorithme une seule fois pour qu'il fonctionne avec n'importe quel type d'image.

Finalement, il est admis qu'il existe une règle concernant les trois points suivants : la généricité, l'efficacité et la facilité d'utilisation. La règle énonce que l'on ne peut avoir que deux de ces avantages qu'en sacrifiant le troisième. Si l'on veut être générique et efficace, alors la solution naïve sera très complexe à utiliser avec beaucoup de paramètres. Si l'on veut qu'une solution soit générique et facile à utiliser, alors elle ne sera pas très efficace par défaut. Enfin, si l'on souhaite qu'une solution soit simple d'utilisation et efficace alors elle ne sera pas très générique. Pour illustrer cette règle, nous pouvons trouver des exemples parmi les bibliothèques existantes. Une bibliothèque notablement générique et efficace en C++ est Boost [169] : elle est également notoirement connue pour être difficile à utiliser. Les composants tels que Boost.Graph, Boost.Fusion ou Boost.Spirit sont difficiles à utiliser. Aussi, une bibliothèque qui est générique et facile à utiliser est le parser Json écrit par Niels Lohmann [172]. Il s'efforce de gérer chaque cas d'utilisation tout en restant très simple à intégrer et à utiliser côté code utilisateur (syntaxe très proche du Json natif en code C++ en fournissant un DSL (Domain Specific Language) [32] pour convertir directement les structures C++ en donnée Json). Cependant, cela a un coût et ce parser est plus lent qu'un parser Json optimisé pour la performance tel que simdjson [171] dont le but est de « parser des gigaoctets de JSON par seconde ». Enfin, il existe de nombreux exemples de code convivial et efficace qui ne sont pas génériques. Nous pouvons citer Scikit-image [120] et OpenCV [29], faciles à utiliser et efficace (beaucoup de code SIMD/GPU écrit à la main) mais pas générique en raison des choix de conception.

Dans cette thèse, nous avons choisi de poursuivre les travaux réalisés sur la bibliothèque de traitement Pylene [140]. Cependant, travailler uniquement sur la bibliothèque C++ restreindrait l'utilisabilité de notre travail et donc son impact. C'est pourquoi nous visons à atteindre les utilisateurs mettant au point des prototypes en fournissant un package qui peut être utilisé par un langage dynamique tel que Python, sans sacrifier les performances. En particulier, nous visons

la disponibilité d'utilisation dans un notebook Jupyter. Un objectif très important pour nous est d'être utilisable en milieu éducatif, ce qui est une force de Python. Dans cette bibliothèque, nous montrons comment être générique et performant, tout en restant facile à utiliser. Ce faisant, nous nous efforçons de casser la règle citée précédemment. Le périmètre de la bibliothèque se limite à la morphologie mathématique [111, 83] et la fourniture de types d'image versatiles. Nous tirons parti du langage C++ moderne et de ses nombreuses nouvelles fonctionnalités liées à la généricité et à la performance pour dépasser cette règle dans la zone de traitement d'images. Enfin, nous tentons, d'apporter les outils et les concepts de bas niveau du monde statique (moment de la compilation) au monde du prototypage de haut niveau et dynamique (moment de l'exécution) pour une meilleure diffusion et facilité d'utilisation, grâce un pont entre ces deux mondes.

C'est avec cette philosophie à l'esprit que ce manuscrit présente notre travail de thèse lié au langage C++ appliqué au traitement d'images. Il est organisé comme suit :

Programmation générique (généricité) Nous présentons l'état de l'art sur la notion de généricité. Nous expliquons son origine, comment il a évolué au fil du temps (en particulier dans le langage C++), quels problèmes il résout et quels problèmes il crée. Nous expliquons pourquoi le traitement d'images et la généricité fonctionnent bien ensemble. Enfin, nous faisons le tour des outils existants qui permettent à la généricité (intrinsèquement restreinte au langage compilé) d'exister dans le monde dynamique (avec langages interprétés tels que Python).

Taxonomie pour le traitement d'images : types d'image et algorithmes. Nous présentons notre première contribution dans le domaine du traitement d'images qui consiste à réaliser une taxonomie complète des différentes familles d'images ainsi que des différentes familles d'algorithmes. Nous expliquons, entre autres, la notion de concept et son application au domaine du traitement d'images. Nous expliquons comment extraire un concept d'un code existant et comment l'exploiter pour le rendre plus efficace et plus lisible. Nous proposons enfin notre point de vue sous la forme d'une collection de concepts liés au domaine du traitement d'images.

Les Vues d'Image Nous présentons notre deuxième contribution qui est une généralisation du concept de Vues (tiré du langage C++, du travail sur les *ranges* [143]) aux images. Cela permet la création d'images légères et peu coûteuses à copier. Cela permet également d'avoir une approche beaucoup plus simple pour concevoir une pipeline de traitement d'images en enchaînant les opérations directement dans le code de manière intuitive. Les *ranges* sont le ciment d'une nouvelle façon de designer des algorithmes pour faciliter l'utilisation des images, ce qui améliore donc leur aspect générique. Enfin, nous discutons du concept d'évaluation paresseuse et de l'impact des vues sur les performances.

Un pont entre le monde statique et le monde dynamique Nous présentons notre troisième contribution qui est un moyen de donner accès aux fonctionnalités génériques d'un langage compilé (tel que C++) à un langage dynamique (tel que Python) pour faciliter le passage entre la phase de prototypage et la phase de production. En effet, pour parvenir à une communication efficace entre le code dynamique et le binaire de la bibliothèque statique, il faut être en mesure de concilier du code C++ générique dont la généricité est résolue au moment de la compilation (ce que nous appelons le « monde statique »), et du code Python dynamique qui s'appuie sur des packages binaires pré-compilés (ce que nous appelons le « monde dynamique ») : et cela n'est vraiment pas évident. D'autant plus que nous ne pouvons pas non plus exiger de l'utilisateur qu'il fournisse et utilise un compilateur à chaque fois qu'il veut utiliser notre bibliothèque depuis Python. Nous discutons quelles sont les solutions existantes qui peuvent être envisagées ainsi que leurs avantages et inconvénients. Enfin, nous discutons de la manière dont nous avons conçu et réalisé une solution hybride pour arriver à faire ce pont entre le monde statique et le monde dynamique.

Programmation générique (généricité)

En langage naturel, quelque chose est générique quand il peut répondre à plusieurs objectifs à la fois, tout en étant un minimum efficace. Par exemple, un ordinateur est un outil générique qui permet de rédiger des documents, d'accéder à des e-mails, de parcourir Internet, de jouer à des jeux vidéo, de regarder des films, de lire des e-books etc. En programmation, un outil est dit générique lorsqu'il peut répondre à plusieurs objectifs. Par exemple, le compilateur gcc peut compiler plusieurs langages de programmation (C, C++, Objective-C, Objective-C++, Fortran, Ada, D, Go et BRIG (HSAIL)) que cibler plusieurs architectures (IA-32 (x86), x86–64, ARM, SPARC, etc.). Désormais, on peut dire que gcc est un compilateur générique. À ce stade, il est important de noter que même si un outil est considéré comme générique, il y a une limitation quant à ce que cet outil peut faire et ce qu'il ne peut pas faire. Un compilateur malgré la prise en charge de nombreuses langues et architectures, ne pourra pas passer un appel téléphonique ou faire un café. De ce fait, il est important de noter que la généricité est un aspect qui qualifie quelque chose. Nous étudions les aspects génériques liés aux bibliothèques et aux langages de programmation.

Cette thèse laisse volontairement de côté l'aspect générique lié à l'architecture cible. En effet, savoir écrire et/ou générer du code capable de s'exécuter sur un large éventail d'architectures matérielles différentes est un domaine de recherche à lui tout seul et n'est pas l'objet principal de cette thèse. Il est également connu sous le nom d' *informatique hétérogène*. Au lieu de cela, nous allons nous concentrer sur les aspects liés à la généricité au niveau de la bibliothèque et au niveau du langage de programmation.

Généricité au sein des bibliothèques Elle est décrite par la cardinalité du nombre de cas d'utilisation qu'elle peut gérer. Les bibliothèques fournissent toujours leurs propres structures de données, pour représenter et donner un sens aux données que l'utilisateur veut traiter, ainsi que des algorithmes pour traiter ces données et fournir différents types de résultats. Une bibliothèque sera alors cataloguée comme *générique* [15] lorsque (i) ses structures de données permettent à l'utilisateur de s'exprimer entièrement, sans limitation et lorsque (ii) sa banque d'algorithmes est suffisamment grande pour faire tout ce que l'utilisateur voudrait faire avec ses données. En réalité, une telle bibliothèque n'existe pas et il y a toujours des limitations. Étudier ces limites et quelles raisons les motivent est la clé pour comprendre comment les dépasser à l'avenir, en développant le support de nouveaux matériels et/ou logiciels pour de nouvelles fonctionnalités permettant plus de généricité.

Généricité dans les langages de programmation Elle est décrite par la capacité du langage à exécuter le même code sur une grande quantité de structures de données [31], qu'elles soient natives (char, int, ...) ou définies par l'utilisateur. Il est aujourd'hui primordial qu'un langage de programmation puisse le faire. En effet, dans un monde où les technologies de l'information sont omniprésentes, la quantité de code écrit par les développeurs de logiciels est faramineuse. Et il en va de même pour la quantité de bogues et de vulnérabilités de sécurité. Pouvoir avoir nativement un langage de programmation qui permet de faire *plus* en écrivant *moins* se traduit mathématiquement par un coût de développement et de maintenance réduit. Les langages de programmation offrent de nombreuses façons d'atteindre la généricité qui dépendent des spécificités intrinsèques des langages : compilés ou interprétés, natifs ou émulés, etc.

Avant d'entrer dans les détails de ce que la généricité implique pour les bibliothèques et les langages de programmation, il est nécessaire d'introduire un peu de vocabulaire. Le premier terme est la notion de *type*. Un *type* (ou *type de données*) est un attribut de données qui indique au compilateur ou à l'interpréteur comment le programmeur a l'intention d'utiliser les données. La grande majorité des langages de programmation prend en charge les types de données de base (également appelés types primitifs) tels que les nombres entiers, les nombres à virgules flottantes, les types booléens et les chaînes de caractères (ASCII, Unicode, etc.). Cet attribut de type définit

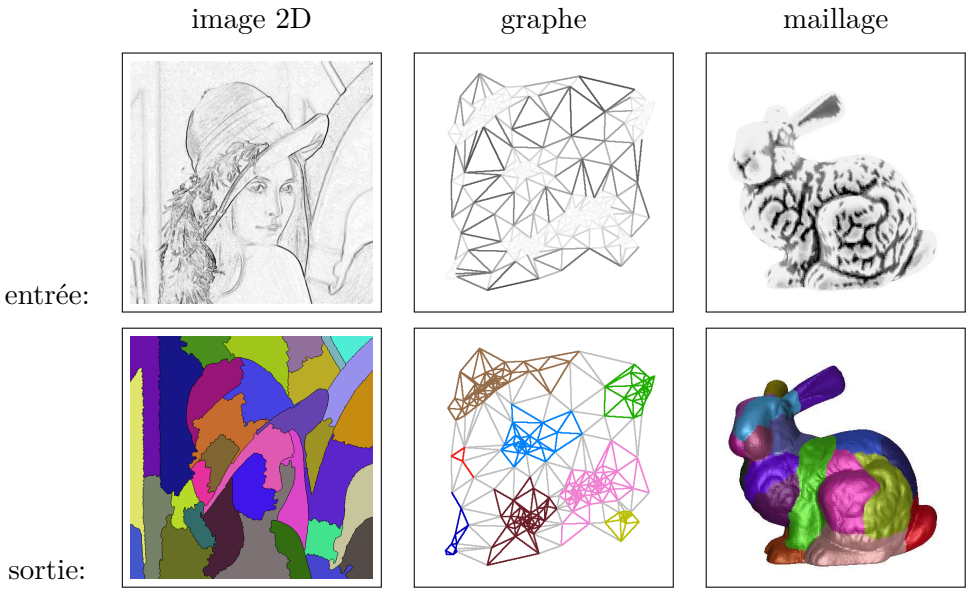
les opérations qui peuvent être effectuées sur les données, la signification des données ainsi que la taille des données en mémoire (les données peuvent alors être stockées sur le tas, la pile, etc.). Un type de données fournit un ensemble de valeurs à partir desquelles une expression (c'est-à-dire variable, fonction, etc.) peut prendre ces valeurs. Parmi les langages de programmation, nous pouvons distinguer ceux qui sont dynamiquement typés et ceux qui sont statiquement typés. Les langages typés statiquement sont ceux dont les variables sont déclarées contenant un type spécifique. Cette variable ne peut contenir des données d'un autre type dans le champ où elle est déclarée. Des langages de programmation à typage statique sont Ada, C, C++, Java, Rust, Go et Scala. Les langages typés dynamiquement sont ceux dont les variables peuvent être réassignées avec une valeur de type différent de celui avec lequel elle a été initialement déclarée. Le type de variable est ensuite modifié dynamiquement pour s'adapter à la nouvelle valeur qu'elle porte. Des langages de programmation à typage dynamique sont PHP, Python, JavaScript et Perl.

La conséquence de pouvoir dire quel type une variable contient à tout moment (langage à typage statique) est double. Pour le développeur, il est plus facile de raisonner sur le code et de repérer les bugs. Pour le compilateur, il est possible de générer un code binaire optimisé spécifiquement pour ce type de données (vectorisation, etc.). Le fait de pouvoir transformer le type qu'une variable peut contenir au moment de l'exécution sert principalement à faciliter le prototypage. Lors de la modification d'un notebook Jupyter, il est très apprécié de ne pas se limiter à un seul type pour chaque variable déclarée afin de pouvoir itérer sur le prototype plus rapidement.

En traitement d'images, une image Im est définie sur un domaine $\mathcal{D}$ (qui contient des points) par la relation $\forall x \in \mathcal{D}, y = Im(x)$ où y est la valeur de l'image Im pour le point x. Cette définition se traduit toujours par une structure de données complexe lorsqu'elle est transposée dans un langage de programmation. Cette structure de données doit connaître la mémoire tampon contenant les données de l'image que des informations sur la taille et les dimensions de l'image. De plus, ajoutant à la difficulté, les informations nécessaires pour définir précisément la structure des données ne sont pas toujours connues lors de l'écriture du code source. En effet, un cas d'utilisation très simple consiste à lire une image dans un fichier pour la charger en mémoire. Le fichier peut contenir une image de différents types de données et le programme doit toujours fonctionner correctement. Il y a de multiples approches pour résoudre ce problème, que nous allons aborder.

En projetant la notion de généricité au traitement d'images, nous pouvons déduire que nous avons besoin de deux aspects importants pour être générique. Tout d'abord, nous devons décorréler les structures de données de leur topologie et des données sous-jacentes des algorithmes. En effet, nous voulons que nos algorithmes supportent autant de structures de données que possible. Ensuite, nous pouvons factoriser ensemble de nombreux algorithmes partageant le même schéma calculatoire.

La généricité peut avoir deux significations différentes selon les personnes que vous interrogez. Par exemple, certaines diront que la généricité est un aspect haut niveau et vont qualifier un outil comme étant « suffisamment générique » lorsque celui-ci peut gérer tous ses cas d'utilisation. D'autres soutiendront que la généricité est un aspect bas niveau et va donc concerner la machine (code) fabriquant les outils, « suffisamment générique » pour fabriquer toutes sortes d'outils. Ni l'un ni l'autre n'a tort. Cependant, pour des raisons de compréhension, nous utiliserons des mots différents pour chacun de ces cas. Un outil assez générique pour gérer un grand nombre de cas d'utilisation sera appelé *versatile*. Enfin, pour un outil dont le but est d'être capable de fabriquer un grand nombre d'outils différents (c'est-à-dire fournir un environnement de programmation capable de gérer n'importe quel cas d'utilisation), nous utiliserons le terme *générique*. Dans cette thèse, la généricité concernera le code, donc l'environnement de programmation capable de gérer n'importe quel cas d'utilisation. La fig. 1 [92] illustre le résultat d'une même implémentation générique de l'algorithme de ligne de partage des eaux, qu'il soit appliqué sur une image 2D, un graphe ou un maillage.



Le même code tourne sur toutes ces différentes données d'entrée.

Figure 1: Algorithme de ligne de partage des eaux appliqué à trois types d'image différents [118].

Nous présentons l'origine de la programmation générique, qui remonte à l'année 1988, year et comment elle a évolué pour être intégrée dans le langage de programmation Ada puis dans le langage de programmation C++. Son évolution s'est poursuivie avec l'arrivée de la notion de *concept* qui complétera la boîte à outils nécessaire pour pouvoir pleinement utiliser la programmation générique sans recourir à des techniques et outils obscurs.

Nous explorons les possibilités de réaliser de la programmation générique au sein d'une bibliothèque. En effet, il y a trois techniques permettant à l'utilisateur d'écrire en une seule fois un algorithme de haut niveau pouvant s'exécuter sur tous les types. Il s'agit des approches de *duplication de code*, de *généralisation* et de *polymorphisme d'inclusion et paramétrique* [72].

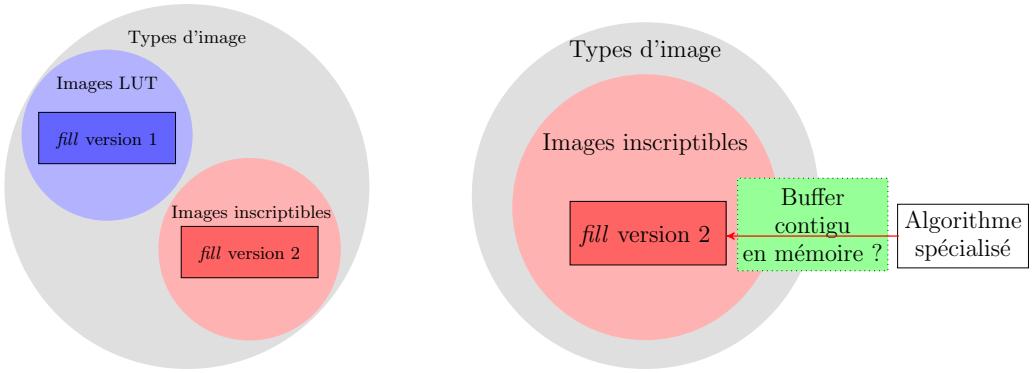
Enfin, nous présentons la limitation inhérente aux templates C++, à savoir qu'ils appartiennent au monde statique (moment de la compilation). La généricité (au sens concept C++) n'existe pas dans le binaire final livré à l'utilisateur. L'utilisateur final, dans son monde dynamique (moment de l'exécution) ne peut pas utiliser un outil générique (code C++). Nous discutons des différentes approches possibles pour combler cet écart entre le monde statique (à la compilation) et dynamique (à l'exécution).

Par la suite nous ferons un large usage de la généricité pour présenter la première contribution de cette thèse : une taxonomie des concepts liée au traitement d'images.

Taxonomie pour le traitement d'images : types d'image et algorithmes

Dans cette thèse, nous avons recherché la meilleure façon d'appliquer les nouvelles fonctionnalités génériques du langage C++ dans le domaine du traitement d'images. Cela nous permet de les tester de manière pratique sur notre zone de prédilection tout en nous souvenant de nos travaux passés, à la fois les succès et les échecs en la matière. Cependant, faire naître des concepts à partir du code est un procédé qui se fait de manière émergente. En conséquence, les premiers travaux sont de faire un inventaire de tous les algorithmes d'images existants ainsi que de tous les algorithmes de traitement d'images (à la fois les plus basiques comme les plus complexes) auxquels nous pouvons penser. De cette façon, nous remarquons des modèles de comportement émergeant de types d'image similaires ou algorithmes similaires. Nous pouvons alors extraire des schémas comportementaux de cet inventaire afin de produire une taxonomie complète sous la forme d'un environnement constitué de concepts liés au traitement d'images.

Nous présentons que les concepts ne sont pas conçus d'après des structures de données, mais d'après des algorithmes. En effet, un concept consiste à extraire un schéma comportemental cohérent d'un bout de code (algorithme) et à le nommer pour lui donner une signification. À travers un exemple simple, mais concret, nous présentons de manière didactique comment extraire des concepts d'un algorithme de traitement d'images (correction gamma).



(a) Differentes versions de l'algorithm *fill*

(b) Specialisation existant au sein d'une version

Figure 2: Ensemble des versions d'algorithme (a) et de ses spécialisations existant au sein d'une version (b).

Nous expliquons ensuite comment, en théorie, les types d'image sont reliés les uns aux autres. Nous présentons l'ensemble de différents types d'image et comment les algorithmes existent dans ces ensembles, ce qui introduit la notion de *version* d'un algorithme. Un algorithme aura différentes *versions* pour chaque ensemble de types d'images qu'il prend en charge. Nous le distinguons (dans la fig. 2) des *spécialisations* d'un algorithme, ces dernières étant la possibilité de profiter d'une opportunité (liée à une propriété) pour faire une optimisation et augmenter les performances.

Nous décrivons ensuite la notion de canevas d'algorithme qui est le résultat découlant de la taxonomie des algorithmes de traitement d'images. En effet, il existe trois grandes familles d'algorithmes : les algorithmes fonctionnant pixel par pixel (par exemple, binarisation), les algorithmes locaux (par exemple, dilatation) et les algorithmes globaux (par exemple, transformée de distance de Chamfer). Nous nous concentrons principalement sur les algorithmes locaux et comment ils peuvent tous être écrits à travers le même canevas de code. En effet, par exemple, la seule différence entre une dilatation et une érosion est l'opérateur (max vs. min). Nous discutons ensuite des possibilités offertes par l'exploitation de ces canevas pour possiblement résoudre des problèmes informatiques hétérogènes.

Enfin, nous introduisons notre première contribution principale : une taxonomie complète relative au domaine du traitement d'images. Nous introduisons d'abord des concepts fondamentaux tels que *point*, *pixel*, *domaine* et *image* (illustrés dans la fig. 3). Nous motivons et introduisons ensuite des concepts avancés liés aux images et aux différentes manières d'accéder aux données (parcours en avant, renversé, indexation, accès direct à la mémoire tampon sous-jacente, ...). Pour finir, nous introduisons les concepts liés aux notions gravitant autour du traitement d'images, telles que les *éléments structurants*, les *voisinages* et les *extensions* (gestion des bordures) qui sont nécessaires pour pouvoir travailler avec des algorithmes locaux.

Par la suite, nous utilisons les concepts présentés pour introduire la deuxième contribution principale de cette thèse : les *vues d'image*.

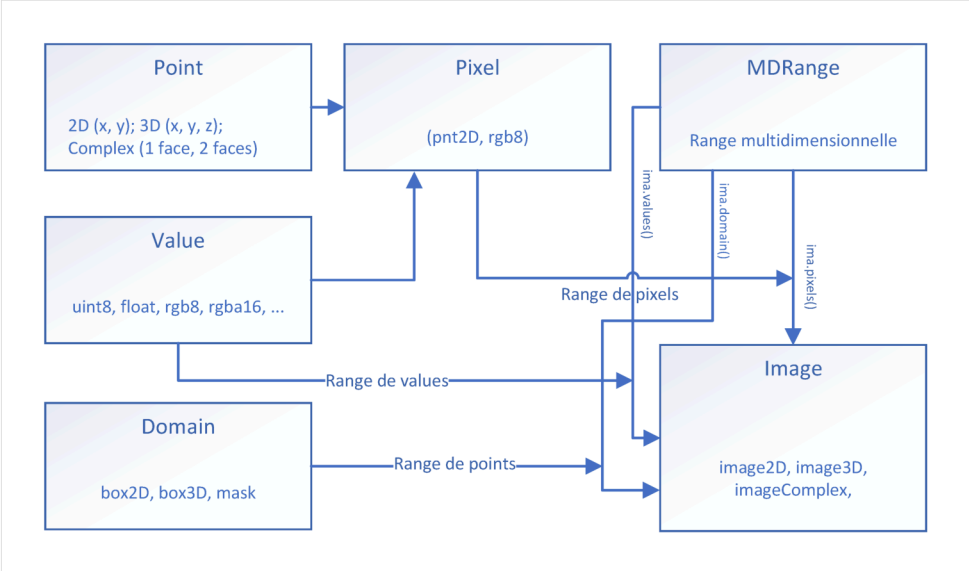


Figure 3: Concept Image.

Les Vues d'Image

Cette notion de vues n'est pas nouvelle [24] et est apparue naturellement en Traitement d'images avec Milena [96, 85] sous le nom de *morpher* [80, 95]. Il était toujours utile de pouvoir projeter une image à travers un prisme qui pourrait extraire des informations spécifiques à son sujet sans avoir besoin de copier la mémoire tampon des données sous-jacentes. Aujourd'hui (2020), le langage C++ (norme 20) introduit aussi ce mécanisme avec les ranges [184] pour les *collections non-propriétaires* (de leur mémoire). Il est nommé *vues* (view) et permet à l'utilisateur d'accéder au contenu d'une collection de données (vector, map) à travers un prisme. Dans Pylene, nous avons décidé de nous aligner sur la nomenclature normalisée dans C++20 afin de ne pas dérouter l'utilisateur. De cette façon, une vue `transform` dans le traitement d'images fait la même chose sur une image que ce que la vue `transform` fait sur un conteneur (collection) dans la bibliothèque de ranges standard. Les *vues* présentent les propriétés suivantes : *copie quasi gratuite*, *non-propriétaire* (ne *possède* aucune mémoire tampon contenant des données), *évaluation paresseuse* (l'accès à la valeur d'un pixel peut nécessiter des calculs) et *composable*. Lorsque les vues sont chaînées, le compilateur construit un *arbre d'expressions* (ou *expression template* tel qu'utilisé dans de nombreuses bibliothèques de calcul scientifique telles qu'Eigen [84]). Le compilateur connaît donc le type de la composition finale et s'assure qu'il n'y ait pas de surcoût en performance à l'exécution (0-overhead).

En traitement d'images, un algorithme s'écrit naïvement en prenant une ou plusieurs données d'entrée (parmi lesquelles figurent la/les image(s) d'entrée), en effectuant un traitement sur ces données d'entrée puis en retournant les données résultantes (ou une erreur). Prenons par exemple l'algorithme mélange alpha (alpha-blending) qui peut être implémenté en C++ naïf comme suit :

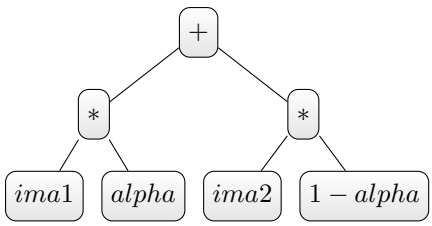
```
void blend_inplace(const uint8_t* ima1, uint8_t* ima2, float alpha,
int width, int height, int stride1, int stride2) {
  for (int y = 0; y < height; ++y) {
    const uint8_t* iptr = ima1 + y * stride1;
    uint8_t* optr = ima2 + y * stride2;
    for (int x = 0; x < width; ++x)
      optr[x] = iptr[x]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

Ce code a plusieurs défauts. Il fait des hypothèses fortes sur les images d'entrée : la contiguïté de ses données dans sa mémoire tampon et sa forme (2D). Supposons que notre utilisateur veuille maintenant restreindre l'algorithme à une région spécifique à l'intérieur de l'image. Le

mainteneur devrait alors fournir une surcharge de fonction pour l'algorithme avec un argument d'entrée supplémentaire correspondant à la région d'intérêt. Supposons que l'utilisateur veuille ensuite prendre en charge la manipulation d'images 3D. Le mainteneur devrait également fournir deux surcharges de fonction supplémentaires avec un argument de *pas* supplémentaire (un pour l'algorithme de base, un l'algorithme prenant en compte une région d'intérêt). Supposons finalement que l'utilisateur souhaite uniquement manipuler le canal de couleur rouge. Alors le mainteneur doit le prendre en charge et ajouter des surcharges de fonctions supplémentaires pour chaque canal et/ou type de couleurs gérés. La complexité augmente grandement pour chaque point de customisation que le mainteneur souhaite offrir à l'utilisateur. Bien sûr, il est possible d'empêcher la duplication de code grâce à une utilisation intelligente des techniques d'ingénierie informatique (factorisation de code, etc.) mais la complexité fuiterait toujours à travers l'API dans une certaine mesure. C'est ainsi que l'autre solution consiste à permettre à l'utilisateur d'effectuer ces restrictions en amont de l'algorithme de manière transparente afin que l'algorithme en aval soit facile à écrire, comprendre et maintenir. Pour y parvenir, nous devons augmenter le niveau d'abstraction autour des images d'un niveau afin que nous puissions travailler directement au niveau de l'image. L'algorithme de mélange alpha (alpha-blending) s'écrirait alors comme indiqué dans la fig. 4.

Figure 4: Algorithme mélange alpha (alpha-blending) écrit au niveau de l'image.

Cette façon d'exprimer un algorithme est obtenue en introduisant des *vues* dans le traitement d'images. Une image est maintenant une vue et peut être restreinte/projetée/manipulée selon les besoins de l'utilisateur avant de la transmettre à un algorithme. Même l'algorithme de mélange alpha (alpha-blending) peut entièrement être réécrit en termes de vues, comme montré dans la fig. 5.




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

Figure 5: Mélange alpha (alpha-blending), implémentation générique avec les *views*, et son arbre d'expressions.

Être capable d'effectuer de puissantes manipulations sur les images avant de les passer aux algorithmes annule complètement le problème initial qui consiste à avoir plusieurs surcharges de fonctions pour un même algorithme tout en maintenant et en documentant tous les arguments optionnels associés. En effet, pour effectuer la transformation de mélange alpha (alpha-blending) sur l'image d'entrée, tout ce que l'utilisateur doit faire est :

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

Si l'utilisateur souhaite restreindre la région à mélanger ou le canal de couleur sur lequel travailler, il lui suffit d'écrire la modification suivante :

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

La restriction est faite en amont de l'algorithme et propagée en aval sans augmenter la complexité du code. De cette façon, les vues augmentent considérablement ce que l'utilisateur peut faire, tout en écrivant moins de code.

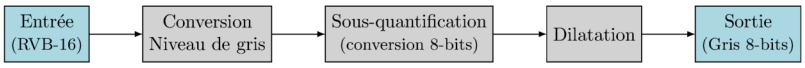


Figure 6: Exemple d'une pipeline de traitement d'images simple.

Nous voyons que *les vues sont composables*. L'une des caractéristiques les plus importantes dans la conception d'une pipeline (généralement, en génie logiciel) est la *composition d'objets*. Elle permet de composer des blocs simples entre eux pour former des blocs complexes. Ces blocs complexes peuvent alors être gérés comme s'il s'agissait à nouveau de blocs simples. Dans la fig. 6, nous avons 3 opérateurs de traitement d'image simples *Image* → *Image* (la conversion en niveaux de gris, la sous-quantification et la dilatation). Comme indiqué dans la fig. 7, la composition des algorithmes considérerait ces 3 opérateurs simples comme un seul opérateur complexe *Image* → *Image* qui pourrait ensuite être utilisé dans une pipeline de traitement d'image encore plus complexe. Tout comme les algorithmes, les vues d'image sont composables. Par exemple, une vue de la vue d'une image reste toujours une image. Dans la fig. 7, nous composons l'image d'entrée avec une vue de transformation en niveaux de gris puis avec une vue de sous-quantification qui alimente enfin l'algorithme de dilatation.

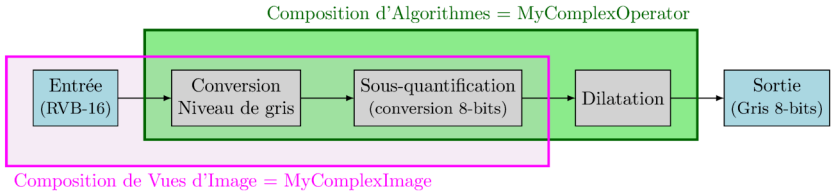


Figure 7: Exemple d'une pipeline de traitement d'images simple illustrant la différence entre la composition d'algorithmes et la composition de vues d'image.

Les *vues améliorent également l'utilisabilité*. Le code pour composer des images dans la fig. 7 est presque aussi simple que :

```
auto input = imread(...);
auto A = transform(input, [](rgb16 x) -> float {
  return (x.r + x.g + x.b) / 3.f; });
auto MyComplexImage = transform(A, [](float x)
  -> uint8_t { return (x / 256 + .5f); });
```

Les personnes familiarisées avec la programmation fonctionnelle peuvent remarquer des similitudes avec ces langages où *transform* (*map*) et *filter* sont des opérateurs de séquence. Les vues utilisent le paradigme fonctionnel et sont créées par des fonctions qui prennent une fonction en argument : l'opérateur ou le prédicat à appliquer pour chaque pixel ; nous n'itérons pas à la main sur les pixels de l'image.

De plus, les *vues améliorent la ré-utilisabilité*. Les extraits de code ci-dessus sont simples, mais peu réutilisables. Cependant, suivant le paradigme de la programmation fonctionnelle, il est assez facile de définir de nouvelles vues, car certains adaptateurs d'image peuvent être considérés comme des *fonctions d'ordre supérieur* pour lesquelles nous pouvons lier certains paramètres, comme nous le ferions avec la technique de currying [17]. Dans la fig. 8, nous montrons comment la primitive *transform* peut être utilisée pour créer une vue additionnant deux images, une vue effectuant la conversion en niveaux de gris et une vue effectuant une sous-quantification. Ces vues

```
auto operator+(Image A, Image B) {
  return transform(A, B, std::plus<>());
}
auto togray = [](Image A) { return transform(A, [](auto x)
  { return (x.r + x.g + x.b) / 3.f; };)
};
auto subquantize16to8b = [](Image A) { return transform(A,
  [](float x) { return uint8_t(x / 256 +.5f); });
};

auto input = imread(...);
auto MyComplexImage = subquantize16to8b(togray(A));
```

Figure 8: Utilisation de fonctions d'ordre supérieur pour créer des opérateurs de vues personnalisées.

de base, réutilisable par la suite sont ensuite chaînées ensemble pour créer une image complexe. (Ces fonctions auraient pu être écrites de manière plus générique pour plus de ré-utilisabilité, mais ce n'est pas le but ici.)

Par ailleurs, *les vues sont évaluées paresseusement.* L'opération étant enregistrée dans la vue d'image, ce nouveau type d'image permet de mélanger des types d'image fondamentaux avec des algorithmes. Dans la fig. 8, la création de vues n'implique aucun calcul en soi, mais retarde plutôt le calcul jusqu'à ce que l'expression `v(p)` soit invoquée. Parce que les vues peuvent être composées, l'évaluation peut être assez retardée. Les adaptateurs d'image sont des *expression template* [20, 40] car ils enregistrent les *expressions* utilisées pour générer l'image en tant que paramètre template. Une vue représente en fait un arbre d'expressions (fig. 5).

En outre, *les vues sont efficaces.* Avec un design classique, chaque opération de la pipeline est implémentée « par elle-même ». Chaque opération nécessite que de la mémoire lui soit allouée pour l'image de sortie. De même, chaque opération nécessite que l'image soit entièrement parcourue. Ce design est simple, flexible, composable, mais n'est pas efficace ni en termes de mémoire ni en performance de calcul. Avec l'évaluation paresseuse, l'image n'est parcourue qu'une seule fois (lorsque la dilatation est appliquée), ce qui a deux avantages. Premièrement, il n'y a pas d'images intermédiaires, ce qui est très efficient en termes de mémoire. Deuxièmement, le parcours de l'image est plus rapide grâce à une meilleure utilisation du cache mémoire et une traversée sélective optimale. En effet, dans notre exemple (fig. 6), traiter un pixel RVB16 depuis l'algorithme de dilatation le convertit directement en niveaux de gris, puis le sous-quantifie en 8 bits, pour enfin le rendre disponible dans l'algorithme de dilatation. Il agit *comme si* nous écrivions un opérateur optimal qui combinerait toutes ces opérations. Cette approche est quelque peu liée aux opérations de fusion du noyau disponibles dans certaines spécifications HPC [150], mais la fusion de vues est uniquement optimisée par le compilateur C++ [139]. L'aspect sélectif se manifeste lorsqu'une région d'intérêt intervient dans la pipeline de traitement. En effet, l'intégralité de la pipeline n'est alors exécutée que sur la région d'intérêt, et cela même si cette sélection n'est faite qu'à la toute fin de la pipeline de traitement.

Enfin, les *vues améliorent la productivité.* Tous les algorithmes de traitement d'images fonctionnant pixel par pixel peuvent (et doivent) être réécrits intuitivement en utilisant une vue en une seule ligne. Les vues *transform* sont la clé permettant ce point. Cela implique qu'il existe un nouveau niveau d'abstraction disponible pour le traiteur d'image lors du prototypage de son algorithme. Le temps passé à la mise en œuvre des fonctionnalités est réduit, donc le temps de la boucle de rétroaction l'est également. Cela amène naturellement un gain de productivité au traiteur d'image.

Un pont entre le monde statique et le monde dynamique

Dans le monde de la programmation, il existe trois grandes familles de langages de programmation [38] qui sont :

1. les langages de programmation *compilés*, tels que C, C++, Rust ou Go,
2. les langages de programmation *interprétés*, tels que Python, PHP, Lisp ou Javascript,
3. les langages de programmation hybrides, tels que Java ou C#.

Ces derniers ont une passe de compilation rapide qui compile le code source dans un bytecode intermédiaire. Ensuite, ce bytecode est interprété via un interpréteur sur une machine hôte.

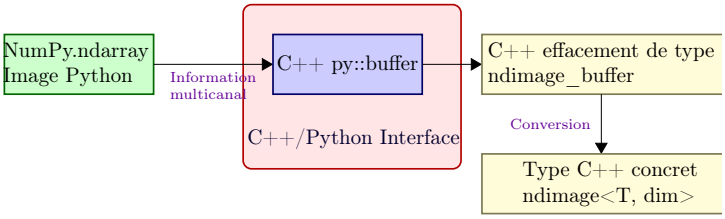


Figure 9: Pont entre Python et C++ grâce à Pybind11 et un effacement de type en C++.

Nous concevons de nombreuses solutions pour résoudre plusieurs types de problèmes liés au pont entre le monde statique et le monde dynamique. Nous présentons notre solution hybride capable de rendre disponible notre bibliothèque générique C++ (statique) à partir de Python (dynamique). Nous discutons à propos des différentes façons de réaliser ce pont, leurs avantages et leurs inconvénients. Aussi, nous introduisons une nouvelle couche d'abstraction, le value-set, qui est un moyen standard de manipuler les valeurs sous-jacentes d'une image, utilisable lors de la mise en œuvre d'algorithmes de traitement d'images. Cette nouvelle couche d'abstraction permet notamment à l'utilisateur d'injecter du code côté Python dans des routines C++ déjà compilées. Cependant, superposer les couches d'abstraction l'une après l'autre, ou même appeler du code Python entraîne forcément un surcoût du côté des performances. C'est pourquoi nous avons réalisé un benchmark pour exposer clairement le coût, en temps d'exécution, de nos différentes solutions. Ce benchmark compare les quatre versions de notre algorithme d'étirement (stretch) dont l'implémentation et l'utilisation sont détaillées dans le manuscrit. Le résultat est affiché dans le table 1.

Type de Dispatch	Compute Time	ΔCompute Time
Value-set natif avec des types de valeurs C++ natifs (baseline)	$0.0093s$	0
Value-set comprenant un appel virtuel avec des types de valeurs C++ natifs	$0.1213s$	×13
Value-set comprenant un appel virtuel avec des types de valeurs C++ cachées par un effacement de type	$1.0738s$	×115
Value-set injecté depuis Python avec Python avec des types de valeurs C++ natifs	$21.5444s$	×2316

Table 1: Benchmark de toutes nos versions de l'algorithme d'étirement (stretch).

Ce benchmark montre que chaque fois qu'une couche d'abstraction est ajoutée au-dessus de la baseline, l'utilisateur doit s'attendre à un facteur 10× impactant les performances de son code. De plus, appeler du code Python est immensément plus lent (2300× !) que la baseline. Cela renouvelle l'intérêt de recompiler la bibliothèque C++ générique avec un type paramétrique supplémentaire connu plutôt que de l'injecter depuis Python, surtout pour du code qui met longtemps à s'exécuter. Pouvoir injecter du code Python facilite le prototypage et augmente la

vitesse à laquelle l'utilisateur peut écrire son code. Cependant, le benchmark montre que ce n'est pas une solution viable une fois que le prototype doit être déployé dans un environnement de production.

Continuité : Solutions basées sur le JIT, avantages et inconvénients Notre solution hybride a certainement des avantages, mais son inconvénient majeur est la lenteur pour injecter nos propres types depuis Python. Il existe une autre solution que cette thèse n'a pas eu l'occasion d'approfondir. Cette solution est basée sur une technologie connue : la compilation Just-In-Time (JIT) qui a été illustrée précédemment dans la fig. 10 (et qui elle-même repose sur la notion de programmation générative [30]). Des bibliothèques telles que AsmJit [91] sont capables d'émettre du code machine directement en effectuant un appel depuis du code C++. En effet, c'est une technologie déjà utilisée par les langages interprétés tels que Java ou PHP pour générer à la volée du code machine natif et optimisé pour la section du code source qui est considéré comme « chaud » par l'interpréteur. Un code source est « chaud » lorsqu'il est beaucoup exécuté : l'utilisateur final gagnerait beaucoup à payer le temps de compilation une fois pour que ce code soit exécuté plus rapidement plusieurs fois par la suite. Appliquer cette solution à notre problématique signifierait que l'utilisateur devrait être capable de compiler du code machine natif à partir du code C++ générique en injectant le type demandé, en tant que paramètre template, lorsqu'il est utilisé. Une telle opération déplace une forte charge sur l'utilisateur, la compilation (habituellement gérée par le mainteneur), et il est bien connu que compiler du code C++ est notoirement *compliqué* et *lent*. De plus, la bibliothèque doit être capable de générer automatiquement l'interface la liant au Python une fois le code C++ compilé et de gérer nativement les types *NumPy.ndarray* dans l'interface. Il existe plusieurs solutions pour parvenir à cet objectif.

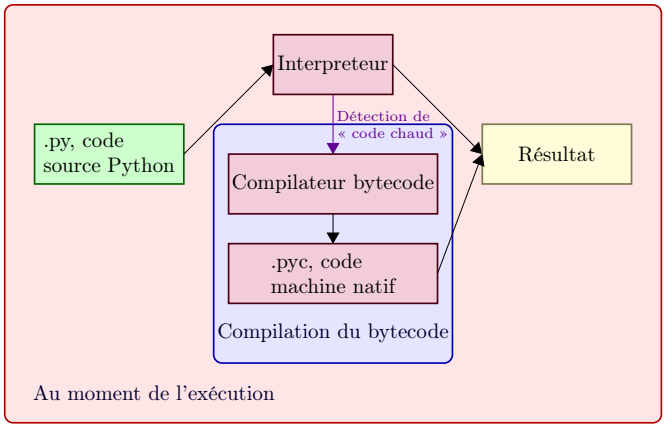


Figure 10: Langage interprété : diagramme d'exécution

La première solution consiste basiquement à faire des appels système aux compilateurs pour réellement *compiler* le code C++ une fois que le/les types template sont connus et explicitement instanciés dans le code source. Cette solution nécessite une génération minutieuse de code, et que l'utilisateur possède un compilateur fonctionnel sur son ordinateur. De plus, l'utilisateur doit résoudre toutes les dépendances de la bibliothèque, comme *freeimage* pour les entrées/sorties, etc. Cette solution a été conçue dans la bibliothèque VCSN [108]. En effet, à chaque fois que l'utilisateur déclare un nouvel automate dans son notebook Jupyter, le code source correspondant est compilé en arrière-plan puis est mis en cache. C'est une solution très périlleuse à mettre en place lorsque l'environnement d'exécution final (OS, logiciels installés) n'est pas bien connu à l'avance. De nos jours, le problème peut être moindre. Cependant, il nécessite toujours de maintenir à la fois la bibliothèque et la solution de conteneur pour l'utiliser (par exemple Docker).

La deuxième solution est d'utiliser Cython [88]. C'est une infrastructure de transpilation qui transforme un code source Python directement dans du code source du langage C afin qu'il puisse être compilé par un compilateur C/C++ standard simplement en résolvant l'édition des liens contre l'API Python/C. Cela supprime la charge d'écrire la routine de génération minutieuse de code, les appels système au compilateur C++, et cela supprime aussi le besoin de résoudre toutes les dépendances. Cette infrastructure s'occupe de tout pour l'utilisateur. Cython prend même en charge les template C++ [176] qui sont obligatoires pour notre cas d'utilisation.

La troisième solution consiste à s'appuyer sur des projets récents qui reposent tous sur l'infrastructure LLVM. Nous pouvons notamment citer AutoWIG [141], Cppyy [130], Xeus-cling [173] et Pythran [123]. AutoWIG a du code « maison » basé sur LLVM/Clang pour analyser le code C++ afin de générer et compiler une interface Python via Swig à l'aide du moteur de template Mako. AutoWIG, couplé à Cython permettrait à l'utilisateur, par exemple, de générer du code C lié à une structure Python personnalisée. Ensuite, un simple appel à AutoWIG analysera le code C et l'injecterait dans la bibliothèque C++ pour générer l'interface utilisable par l'utilisateur. En ce qui concerne Cppyy, il est basé sur LLVM/Cling, un interpréteur C++, et peut interpréter directement du code C++ à partir d'une chaîne de caractères Python. Cela permet d'injecter facilement des types personnalisés, qu'ils soient dans du code Python (possiblement transpilé avec Cython) ou du code C++ (directement interprété par Cling). Ensuite, l'infrastructure génère l'interface utilisable à partir de la bibliothèque C++ générique pour le type injecté demandé. Xeus-cling est un noyau Jupyter prêt à l'emploi permettant l'utilisation du code C++ directement à partir d'un notebook. Cela rend possible de contourner complètement le besoin d'avoir une interface avec Python et permet à l'utilisateur d'utiliser la bibliothèque depuis un notebook comme s'il s'agissait une bibliothèque Python. Enfin, Pythran est un compilateur « avancé » pour un sous-ensemble du langage Python, axé sur le calcul scientifique. Il prend un code Python annoté avec quelques descriptions dans son interface et le transforme en un module Python natif avec la même interface, mais en espérant qu'il soit plus rapide. Pythran tire parti du multicœur et des instructions SIMD pour transformer son sous-ensemble du langage Python en code C++ fortement générique instancié pour des types optimisés bien particuliers. Toutes ces infrastructures, cependant, ont un coût élevé en termes de taille de binaire. En effet, un compilateur C++ n'est pas petit et l'embarquer avec la bibliothèque de traitement d'images peut facilement avoir un impact considérable sur la taille du binaire final. Sans l'infrastructure LLVM, le binaire peut peser environ 3 Mo. Avec l'infrastructure LLVM, le poids du binaire devient au strict minimum 50 Mo. De plus, ces solutions peuvent ne pas être immédiatement plus rapides. En effet, lors de la phase de prototypage et des multiples allers-retours avec une large variété de types, l'utilisateur peut ne pas vouloir se montrer patient à attendre les longs temps de compilation à chaque fois qu'il teste avec une nouvelle itération de son travail. Malgré cela, ces solutions offrent d'excellentes voies de recherche pour l'avenir et nous sommes impatient de les explorer.

Conclusion

Le travail que nous présentons dans cette thèse suit un arc narratif très clair. L'accent a d'abord été mis sur la présentation de la notion de programmation générique (généricité), son histoire et comment chacun(e) peut la comprendre, notamment dans sa vie de tous les jours, et en particulier lorsqu'elle est appliquée au traitement d'images. La généricité est une notion vieille de quatre décennies qui a évolué et trouvé une utilisation dans des domaines modernes de notre société. En effet, le traitement d'images est largement utilisé pour construire des applications utilisées dans le monde entier. Cependant, il a été démontré à quel point il peut être difficile de mettre en œuvre des solutions reposant sur la généricité. En effet, il existe une règle de trois liant généricité, performance et facilité d'utilisation énoncée en introduction. Dans cette thèse, nous essayons de démontrer comment briser cette règle, et ce, en trois étapes.

La première étape, illustrée dans *Taxonomy for Image Processing*, a consisté à faire un inventaire des types et familles d'images ainsi que des différents algorithmes de d'images. L'objectif a été de produire une taxonomie complète des types (pixel, image, éléments structurants, ...) et algorithmes (locaux, globaux, ...) liés au traitement d'images afin de pouvoir extraire les Concepts (au sens Concepts C++). Cette première étape délimite le périmètre de ce que nous englobons dans la notion de *généricité*. À partir de ce point de départ, il devient plus facile d'écrire des algorithmes de traitement d'images, juste en s'appuyant sur ces concepts. De plus, différents concepts existent pour permettre aux développeurs d'algorithmes d'exploiter des propriétés (décomposabilité des éléments structurants, contiguïté du buffer de l'image, ...) afin d'atteindre des performances maximales. À ce stade, nous raisonnons encore à un niveau bas (pixel) ce qui génère le besoin de designer une couche d'abstraction afin de permettre un prototypage rapide pour des opérations simples tout en garantissant une empreinte mémoire très faible et proche de zéro impact sur les performances.

Pour cette raison, dans cette seconde étape, nous étendons le concept de *vues* du standard C++ (2020) aux images et nous clarifions la notion de *vues d'image*. Nous faisons également le choix de conception d'avoir une image légère, peu coûteuse à copier (données partagées dans le buffer) par défaut afin de fusionner les vues et l'image concrète selon le point de vue l'utilisateur. L'évaluation paresseuse, qui se produit systématiquement lors de l'utilisation des vues permet un gain de performances lors du découpage d'images volumineuses. Dans le cas où l'image est traitée dans son entièreté, nous sommes tout de même capables d'obtenir des performances très satisfaisantes qui restent stables. Nous montrons aussi, à travers des cas d'utilisation concrets tels que les algorithmes fonctionnant pixel par pixel et la gestion des bordures, comment l'utilisation des vues simplifie grandement la façon d'écrire des algorithmes de traitement d'images plus complexes, et efficaces par défaut. Nous discutons enfin des limites de cette approche, avec un accent particulier sur la vitesse de parcours d'une image, qui est un cas d'utilisation obligatoire que nous devons maîtriser.

Dans la troisième de cette thèse, nous avons porté notre attention sur la manière dont il est possible de distribuer ce logiciel à la communauté des traiteurs d'image qui travaille principalement avec Python. Cette dernière contribution concentre ses efforts sur la recherche de la meilleure façon de concevoir un pont statique (C++ template, temps de la compilation) — dynamique (notebook Jupyter Python, moment de l'exécution) pour apporter efficacement ces notions (concepts et vues) au traiteur d'image. Cette dernière contribution explore également ce dilemme et propose de l'aborder avec une solution hybride dont le design et les motivations sont expliqués en profondeur. Cette solution hybride s'appuie sur la technique de l'effacement de type qui offre une compatibilité avec *NumPy.ndarray* [161]. Ce type (effacé) est alors capable de se convertir au sein d'un dispatcher de cardinalité $n \times n$ (dimension et type sous-jacent) dans un type C++ template concret et optimisé. Cette solution explique aussi comment écrire très simplement le code « glue » permettant d'exposer en Python des algorithmes déjà existants (en C++) grâce à une mécanique de dispatch fortement inspirée du standard C++ (`std::visit`, `std::variant`). Le but de cette solution est de regrouper en un seul endroit dans le code tous les types supportés dans les dispatchers pour faciliter la maintenance et n'avoir qu'un travail minimal demandé à la personne qui implémente les algorithmes pour les exposer au Python, tout cela en gardant des performances natives. En effet, aucune copie superflue n'est effectuée grâce à *pybind11* et son *type-caster* : une seule conversion est effectuée depuis le type effacé vers le type natif. Tout le travail qui est fait dans l'algorithme est effectué sur le type natif optimisé. Enfin, cette solution offre un moyen d'injecter des types Python personnalisés dans la bibliothèque C++ générique à des fins de prototypage, grâce à une nouvelle couche d'abstraction, mais au prix de lourdes pertes de performance. L'inconvénient de cette solution est évidemment le « gonflement » du binaire dont la taille s'accroît avec la cardinalité des types paramétriques (nombre de types pris en charge multiplié par le nombre d'algorithmes multiplié par le nombre d'autres données supportées (éléments structurants, label map, etc.)).

Nous concluons cette thèse en proposant de nouvelles pistes de recherche autour du domaine de la compilation Just-In-Time (JIT) pour améliorer encore le pont entre le monde statique et le monde dynamique. Nous pensons que cette piste mérite d'être explorée, surtout avec les outils déjà existants très prometteurs (Xeus-cling, Cppyy, Cython, AutoWIG, Pythran) afin de résoudre le problème d'accroissement de la taille du binaire. En effet, nous ne devrions compiler que ce dont l'utilisateur a besoin, mais le prix d'entrée peut être d'embarquer statiquement un interpréteur C++ (LLVM/cling ?) dans son binaire, ce qui en soi entraînerait une augmentation forfaitaire non négligeable de la taille du binaire. Il peut être possible, cependant, de s'appuyer sur l'infrastructure système de l'utilisateur pour que le mainteneur ne soit pas obligé de distribuer un interpréteur/compilateur C++ en même temps que le binaire de sa bibliothèque de traitement d'images. Il s'agit encore d'un domaine de recherche nouveau et nous souhaiterions vivement l'approfondir pour étudier ce qu'il est possible de réaliser dès aujourd'hui avec ces outils pour la communauté du traitement d'images.

Contents

List of Figures

List of Tables

Part I

Context and History of Generic programming

Chapter 1

Introduction

Outline

NOWADAYS *Computer Vision* and *Image Processing (IP)* are omnipresent in the day-to-day life of the people. It is present each time we pass by a CCTV camera, each time we go to the hospital do an MRI, each time we drive our car and pass in front of a speed camera and each time we use our computer, smartphone or tablet. It cannot be avoided anymore. The systems using this technology are sometimes simple and, sometimes, more complex. Also, the usage made of this technology serves many purposes such as space observation, medical imaging, quality of life improvement, surveillance, control, autonomous system, etc. Henceforth, *Image Processing* has a wide range of research and, despite having a mass of previous of work already contributed to, there are still a lot to explore.

Let us take the example of a modern smartphone application which provides facial recognition in order to recognize people whom are featuring inside a photo. To provide an accurate result, this application will have to do a lot of different processing through several steps. In addition, there are a lot of variables to handle. We can list (non exhaustively) the weather, the light exposition, the resolution, the orientation, the number of person, the localization of the person, the distinction between humans and objects or animals, etc. All of these elements needs to be carefully handled in order to finally recognize the person(s) inside the photo. What the application does not tell you is the complexity of the image processing pipeline behind the scene that, most of the time, cannot even be executed in its entirety on one's device (smartphone, tablet, ...). Indeed, image processing is costly in computing resources and would not meet the time requirement desired by the user if the entire pipeline was executed on the device. Furthermore, for the final part which is "recognize the person on the photo", the application needs to feed the pre-processed photo to a neural network trained beforehand through deep learning techniques in order to give an accurate response. There exists technologies capable of embedding neural network into mobile phone such as MobileNets [135], but it remains limited in terms of operational capabilities. It can, for instance, detect a human being inside a photo but not give the answer about whom this human being is. That is why, accurate neural network system usually are abstracted away in cloud technologies making them available only via Internet. When uploading his image, the user does not imagine the amount of technologies and computing power that will be used to find who appears on the photo.

We now understand that, to build applications that interact with photos or videos nowadays, we need to be able to do accurate, fast and scalable image processing on a multitude of devices (smartphone, tablet, ...). In order to achieve this goal, image processing practitioners need to have two kinds of tools at their disposal. The first one is the prototyping environment that is a toolbox which allow the practitioner to develop, test and improve its application logic. The second one is the production environment which deploys the viable version of the application that was developed by the practitioner. Both environments may not have the same needs. On

one hand, the prototyping environment usually requires a fast feedback loop for testing, an availability of state-of-the-art algorithms and existing software. This way the practitioner can easily build on top of them and be fast enough so that he does not wait a long time to get the results when testing many prototypes. On the other hand, the production environment must be stable, resilient, fast and scalable.

When looking at standards in the industry nowadays, we notice that the *Python* programming language is the main choice for prototyping. However, Python may not be suitable to push a viable prototype in production with minimal changes afterwards. We find it non-ideal that the practitioner cannot take advantages of many optimization opportunities, both in terms of better algorithm efficiency and better hardware usage, when proceeding this way. It would be much more efficient to have basic low level building blocks that can be adapted to fit as mush use cases as possible. This way, the practitioner can easily build on top of them when designing his application. We distinguish two kinds of use cases. The first one is about the multiplicity of types or algorithms the practitioner is facing. The second one is about the diversity of hardware the practitioner may want to run his program on. The goal is to have building blocks that can be intelligent enough to take advantage of many optimization opportunities, with regard to both input data types/algorithms and target hardware. Then the practitioner would have an important performance improvement, by default, without specifically tweaking his application. As such, the concept of genericity is introduced. It aims at providing a common ground about how an image should behave when passed to basic algorithms needed for complex applications. This way, in theory, one only needs to write the algorithm once for it to work with any given kind of image.

Different data types and algorithms

In Image Processing, there exists a multitude of image types whose characteristics can be vastly different from one another. This large specter is also resulting from the large domain of application of image processing. For instance, when considering photography we have 2D image whose values can vary from 8 bits grayscale to multiple band 32-bits color scheme storing information about the non-visible specter of human eye. If we consider another domain of application, such as medical imaging, we can now consider sequence of images such as sequence of 3D image for an MRI for instance. More broadly there are two orthogonal constituents of an image: its topology (or structure) and its values. However, there are two more aspects to consider here. Firstly, image processing provide plenty of algorithms that can or cannot operate over specific data types. There are also different kind of algorithms. Some will extract information, (e.g. histogram) others will transform the image point-wise (e.g. thresholding), and some other will even combine several images to render a different kind of information (e.g. background subtraction). There are many simple algorithms and also many complex algorithms out there. Secondly, there are orbiting data around image types and algorithms that are also very diverse and necessary for their functioning. Indeed, a dilation algorithm will need an additional piece of information: the dilation disc. A thresholding algorithm is given a threshold. A convolution filter requires a convolution matrix to operate. That is why, when considering both image types and algorithms, we need a 3D-chart (illustrated in fig. 1.1) to enumerate all possibilities, where one axis is the image topology, one axis is the color scheme and one axis enumerate the additional data that can be associated to an image.

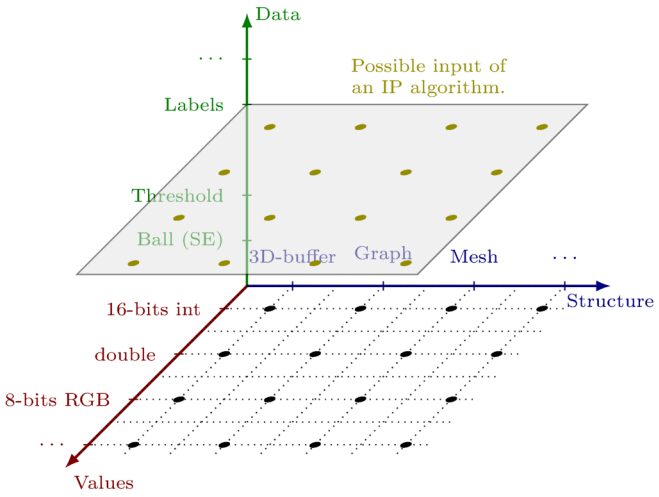


Figure 1.1: Illustration of the specter of the multitude of possibilities in the image processing world.

Different user profiles and their use cases

The end user He is a non-programmer user who wants to occasionally use image processing software through UI-rich interface, such as Adobe Photoshop [167] or The GIMP [156]. His skills are non-relevant as the end user is using the software to get work done even though he does not fully understand the underlying principles. For instance, the end user will want to correct the brightness of an image, or remove some impurities from a face or a landscape. The end user does not want to build an application but wants to save time. His needs mainly revolve around a clean and intuitive software UI as well as a well as support for mainstream image types and operation a photograph needs to do.

The practitioner He is what we become after we first approach the image processing area. A practitioner is the end user of image processing libraries. His skills mainly revolve around applied mathematics for image processing, prototyping and algorithms. A practitioner aims at leveraging the features the libraries can offer him to build his application. For instance, a practitioner can be a researcher in medical imaging, an engineer building a facial recognition application, a data scientist labeling his image sets, etc. The needs of practitioners mainly revolve around a fast feedback loop. The development environment must be easy to set up and access. This way a practitioner can judge quickly whether one library answers his needs. The documentation of the library must be exhaustive and didactic with working examples. When prototyping, the library must provide a fast feedback loop, as in a Python notebook for instance. Finally, the library must be easy to integrate in a standard ecosystem, such as being able to work with *NumPy.ndarray* [75, 69, 161] natively without imposing its own types. To sum up, practitioner's programmatic skills do not need to be high as his main goal since to focus on algorithms and mathematics formulas.

The contributor He is an advanced user of a library who is very comfortable with its inner working, philosophy, aims, strengths and potential shortcomings. As such, he is able to add new specific features to the library and fix some shortcomings or bugs. Usually a contributor is able to add a feature needed by practitioner to finish his application. Furthermore, he can then contribute back this features to the main project via pull requests if it is relevant. This way, a maintainer will assess the pull request and review it. The two main points of a contributor are his deep knowledge of a library and his ability to write code in the same language as the source code of the library. Also, a contributor must have knowledge of programming best practices

such as writing unit tests which are mandatory when adding a feature to an existing library. To facilitate contribution, a library must provide clear contribution guidelines, must be easy to bootstrap and must compile without having heavy requirements or dependencies. The best case would be that the library is handled by standard packages managers such as the system's apt, Python's pip or Conan.

The maintainer He is usually the creator, founder of the library or someone that took over the project when the founder stepped back. Also, when a library grows, it is not rare that regular contributors end up being maintainer as well to help the project. The maintainer is in charge of keeping alive the project by fulfilling several duties, such as upgrading and releasing new features according to the user (practitioner) needs and the library philosophy. Also, a library may not evolve as fast as the user requests it because of lack of time from maintainers. A lot of open source projects are maintained by volunteers and lack of time is usually the main aspect slowing development progress. The maintainer is also in charge of reviewing all the contributors pull requests. He must check if they are relevant and completed enough, (for instance, presence of tests and documentation) to be integrated in the project. Indeed, merging a pull requests equals to accepting to take care of this code in the future too. It means that further upgrade, bug fix, refactoring of the project will consider this new code too. If the maintainer is not able to take care of this code then it should probably not be integrated in the project in the first place. All living projects and libraries have their maintainers. A maintainer is someone very familiar with the inner working and architectural of the project. He is also someone that has some history in the project to understand why some decisions or choices were made at some point in the past, and what the philosophy of the project is. It is important to be able to refuse a contribution that would go contrary to the philosophy of the project, even a very interesting one. Finally, the typical profile of a maintainer is a senior developer that is used to the standard workflow in open source (forks, branches, merge/pull requests and continuous integration).

Different tools

Before stating the topic of the thesis, it is important to enumerate the different kind of tools the market currently has to offer to know where we will be positioning ourselves.

Graphic editors They are what neophyte thinks about when they imagine what image processing is. Those are tools that allow a non-expert user to apply a wide array of operation on an image with an intuitive GUI, in a way the user does not have to understand the underlying logic behind each and every operation he is applying. Such tools are usually large complex software such as The GIMP [156] or Photoshop [167]. Their aim is to be usable by end users while supporting a large set of popular image format and operations.

Command line utilities They are binaries that perform one or more operation, invocable from a console interface or from a shell script through a command line interface (CLI). This CLI usually offers several options to pass data and/or information to the programs in order to perform an action. The information can be, for instance, the input image path, the output image path and the name of a mathematical morphology [111, 83] algorithm to apply. Usually, command line utilities come as projects, such as ImageMagick [174], GraphicsMagick [160] or MegaWave [94, 58].

Visual programming environment They are software that allow the user to graphically and intuitively link one or several image processing operations while interactively displaying the result. The processing can easily be modified, and the results are updated accordingly.

Those pieces of software are usually aimed at engineer or researchers doing prototyping work not exclusive to image processing. Mathcad [154] is a good example of such a software.

Integrated environment They are feature-rich platforms for scientists oriented toward prototyping. Those platforms provide a fully functional programming language and a graphical interface allowing the user to run commands and scripts as well as viewing results and data (image, matrices, etc.). The most well-known integrated environment are Matlab [163], Scilab [165], Octave [170], Mathematica [166] and Jupyter [129] notebooks.

Package for dynamic language It has known a surge in development these last few years and a multitude of libraries has been brought to dynamic languages this way. For instance, let us consider the Python programming language. There are two main package providers: PyPi [175] and Conda [158]. Both allow to install packages to enable the user to program his prototypes in Python very quickly. In image processing, there are packages such as SciPy [45], NumPy, Scikit-image [120], Pillow [168] as well as binding for OpenCV [29].

Programming libraries They are the most common tool available out there. They are a collection of routines, functions and structures providing features through a documentation and binaries. Furthermore, they require the user to be proficient with a certain programming language and also to be able to integrate a library into his project. For image processing we have: IPP [60], ITK [110], Boost.GIL [63], Vigra [37], GrAL [62], DGTal [125], OpenCV [29], CImg [105], Video++ [115], Generic Graphic Library [36] Milena [96, 80, 85] and Olena [191, 92, 95, 118].

Domain Specific Languages (DSL) [32] They are tools developed when a library developer deem he is unable to express the concepts and abstraction layers he wants to express through publishing a library. In this case, the barrier is often the programming language itself and so the developer thinks that another layer of abstraction above the programming language would be a good thing. It leads to the genesis of new programming languages in some cases, like Halide [112] and SYCL [147, 146] but can also be a case of having the current programming language be "upgraded" to include another subset of features that are not natively included. This is often the case in C++ where we have in-language DSL like Eigen [84], Blaze [97, 98], Blitz++ [40, 26] or Armadillo [131]. They leverage a possibility of the C++ programming language (*expression templates* [20]) to achieve it.

Topic of this thesis

In the end, it is often known that there is a rule of three about genericity, efficiency and ease of use. The rule states that one can only have two of those items by sacrificing the third one. If one wants to be generic and efficient, then the naive solution will be very complex to use with lots of parameters. If one wants a solution to be generic and easy to use, then it will be not very efficient by default. If one wants a solution to be easy to use and efficient then it will not be very generic. To illustrate this rule, we can find examples among existing libraries. A notably generic and efficient library in C++ is Boost [169]: it is also notably known to be hard to use. Components such as Boost.Graph, Boost.Fusion or Boost.Spirit are hard to use. Also, a library which is generic and easy to use is the Json parser written by Niels Lohmann [172]. It strives to handle every use case while remaining very easy to integrate and to use in user code (syntax really close to native Json in C++ code by providing DSL to parse C++ constructs into JSON). However, this has a cost and the parser is slower than Json parser optimized for speed such as simdjson [171] whose aim is to "parse gigabytes of JSON per second". Finally, there are plenty of example of user-friendly and efficient code which is not generic. We can cite Scikit-image [120]

and OpenCV [29] that are easy to use and efficient (lot of handwritten SIMD/GPU code) but not generic due to the design choices.

In this thesis, we chose to work on an image processing library through continuing the work on Pylene [140]. But only working at library level would restrict the usability of our work and thus its impact. That is why we aim to reach prototyping users (practitioners) through providing a package that can be used in dynamic language, such as Python, without sacrificing efficiency. In particular, we aim to be usable in a Jupyter notebook. It is a very important goal for us to reach a usability able to permeate into the educational side, which is a strength of Python. In this library, we demonstrate how to achieve genericity and efficiency while remaining easy to use, all at the same time. In doing so, we are endeavoring to break through the rule of three presented previously. The scope of this library is limited to mathematical morphology [111, 83] and to the provision of very versatile image types. We leverage the modern C++ language and its many new features related to genericity and performance to break through this rule in the image processing area. Finally, we attempt, to bring low level tools and concepts from the static world to the high level and dynamic prototyping world for a better diffusion and ease of use, thanks to a bridge between those two worlds.

With this philosophy in mind, this manuscript aims at presenting our thesis work related to the C++ language applied to the Image Processing domain. It is organized as followed:

Generic Programming (genericity) 2 This chapter presents a state-of-the-art overview about the notion of genericity. We explain its origin, how it has evolved (especially within the C++ language), what issues it is solving and what issues it is creating. We explain why image processing and genericity work well together. Finally, we tour around existing facilities that allows genericity (intrinsically restricted to compiled language) to exists in the dynamic world (with interpreted languages such as Python).

Taxonomy for Image Processing: Image types and Algorithms 3 This chapter presents our first contribution in the image processing area which is a comprehensive work consisting in the taxonomy of different image types families as well as different algorithms families. This chapter explains, among others, the notion of *concept* and how it applies to the image processing domain. We explain how to extract a concept from existing code and how to leverage it to make code more efficient and readable. We finally offer our take in the form of a collection of concepts related to image processing area.

Images Views 4 This chapter presents our second contribution which is a generalization of the concept of View (from the C++ language, related to ranges [143]) to images. This allows the creation of lightweight, cheap-to-copy images. It also enables us to design image processing pipeline a much simpler way; simply by chaining operations directly in the code in an intuitive way. Ranges are the cement of new designs to ease the use of image into algorithms which can further extend their generic behavior. Finally, we discuss the concept of lazy evaluation and the impacts of views on performance.

A bridge between the static world and the dynamic world 5 This chapter presents our third contribution which is a way to grant access to the generic facilities of a compiled language (such as C++) to a dynamic language (such as Python) to ease the gap crossing between the prototyping phase and the production phase. Indeed, it is really not obvious to be able to conciliate generic code from C++ whose genericity is resolved at compilation-time (we call it the "static world"), and dynamic code from Python which rely on pre-compiled package binaries (we call it the "dynamic world"), to achieve an efficient communication between the dynamic code and the library. We also cannot ask of the user to provide and use a compiler each time he wants to use our library from Python. In this chapter, we discuss what are the existing solutions that

can be considered as well as their pros. and cons. We then discuss how we designed a hybrid solution to build the bridge between the static world and the dynamic world.

Chapter 2

Generic programming (genericity)

In natural language we say that something is generic when it can fit several purposes at once while remaining decently efficient. For instance, a computer is a generic tool that allows to write documents, access emails, browse Internet, play video games, watch movies, read e-books etc. In programming, we say that a tool is generic when it can fit several purposes. For instance, the gcc compiler can compile several programming languages (C, C++, Objective-C, Objective-C++, Fortran, Ada, D, Go, and BRIG (HSAIL)) as well as target several architectures (IA-32 (x86), x86–64, ARM, SPARC, etc.). Henceforth, we can say that gcc is a generic compiler. At this point it is important to note that, even though a tool is deemed generic, there is a scope on what the tool can do and what the tool cannot do. A compiler, despite supporting many languages and architectures, will not be able to make a phone call or a coffee. As such it is important to note that genericity is an aspect that qualifies something. We will now studies the generic aspects related to libraries and programming languages.

This thesis voluntary leaves out the generic aspect related to the target architecture. Indeed, being able to write and/or generate code that is able to run on a large array of different hardware architectures is a field of research on its own and is not the main focus of this thesis. It is also known as *heterogeneous computing*. This field saw the birth of its own standards (SYCL [147, 146]) and libraries solving different problems, such as Halide [112], which provides its own DSL (Domain Specific Language) to write code that will run on GPUs. In pure C++ there exists several high performance math library for linear algebra, dense and sparse arithmetic which are optimized to produced very optimized code (vectorized instruction support, parallel execution etc.). The most popular libraries are Eigen [84], Blaze [98, 99, 97], Blitz++ [40, 26] and Armadillo [131, 132, 145] leveraging *expression templates* [20] to achieve their goal. Also, we note that Eigen is compatible with GPU source code [177] and can be used inside Cuda kernels. A Cuda extension for Blaze was released recently [152] and allow its use in GPU code as well. Armadillo uses BLAS [46] as underlying linear algebra routines, which enables one to link against the GPU-accelerated NVBLAS (NVidia) [178] or ACML-GPU (AMD) [107] as drop-in replacement for BLAS to offload the work on GPU. All those libraries have set performance as their main goal. They try to provide generic ways to solve issues related to parallelism and/or vectorization while making use of expression templates for lazy computing (which will be seen in section 4.3.3). They do not aim to be able to handle as many input types as possible, however, the lazy-computing techniques is used to generate new types on-the-fly. Henceforth, those libraries still need to embed generic facilities to handle their own internal set of types. This thesis addresses genericity at the input level rather than the target architecture level, henceforth, we will not address this topic here.

History Genericity takes its root in Aug. 1978 when Backus publishes his paper about functional programming [2]. Backus thinks that there exists five computation forms with which one can build up all the rest of the computational infrastructure. Every piece of software, for Backus, is

built from those five functional forms. Furthermore, the initial work of Backus does not use the possibility offered by mutations in his five computational forms. These forms will lead to the birth of the functional programming paradigm (notably famous for its value immutability). Stepanov, a mathematician, thinks that those forms are theorems. He also thinks that there is an infinite number of theorem (as in mathematics) and that reducing their number to five for software programming is reductive. He publishes in 1987 [7] that one cannot ignore the mutability of states if one wants to achieve maximum efficiency. Stepanov reasons about software programming by drawing a parallel with algebraic structures. Indeed, let us consider the classical parallel computation model (map-reduce [77]). In this model, being able to reorder computation is a prerequisite in order to have a reduction that works. *Reordering computation* can be reworded as the *associative property* of an algebraic structure; the monoid [148] which is a triplet consisting of a data structure, an associative binary operation and a neutral element. Stepanov thinks that we extract those data structures and those laws or properties from software program the same way as we discover theorems and axioms in mathematics. The software would then be defined on top of algebraic structures, and it's the software programmer's job to discover the data structures and the laws that compose them.

This reflection leads to the publication of "Generic programming" [8] in which the term *Generic Programming* first appears. "By generic programming, we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parametrized procedural schemata [5] that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.". This article [8] leads to the genesis of the book *The Ada Generic Library linear list processing packages* [9] where was published the first work about a generic library consisting of algorithms and data structures. Then Stepanov and Lee wrote the first version of the Standard Template Library [19] in 1995 which is a carefully crafted library made of basic algorithms to manipulate algebraic structures (data structures) that is still an authority up to this day. This standard template library was then incorporated alongside the C++ language for the release of the first ISO standard of the language in 1998 [25]. That same year is published "Fundamentals of Generic Programming" [31]. This is the first place where the term *concept* appears as "a set of axioms satisfied by a data type and a set of operations on it." This term is designed to include the complexity of an operation as part of an axiom in software programming. Also, it is introduced to replace the previously used mathematical terms that could not carry the notion of complexity. It is also the first place where the notion of *regular* type appears: "Since we wish to extend semantics as well as syntax from built-in types to user types, we introduce the idea of regular type, which matches the built-in type semantics, thereby making our user-defined types behave like built-in types as well." Efforts were made by Gregor et al. in [67, 66] in 2006 to introduce them into C++11 [90], but it ultimately failed, and the feature was pulled off of the C++ standard [186]. This had major consequences on the language. Indeed, the standard body did not publish a standard for 13 years, which is a long period in the information technologies area, leading to adoption of more recent, more maintained/evolving languages by the industry. The standard body then decided to review its publication process and has set a 3 years deadline in between each new standard release. Features must be ready in due date before being merged into the new standard version or else they are delayed to the next standard version (3 years later). The standard body decided that it will not wait for a feature to be ready to publish its next release.

Before publishing *Elements of Programming* [82], Stepanov presented his view of *Generic programming* to Backus. Backus "always knew that at some points he needed to figure out mutation into functional programming and we can view generic programming as functional programming with a well-defined way to handle mutation." Unfortunately Backus passed away before being able to write the forewords of *Elements of Programming* book [144]. The term *generic programming* never appears in this book because Stepanov thought he lost control over

it. In practice, it has evolved into metaprogramming, effectively associated to C++ template metaprogramming instead of being associated with the underlying mathematics, algebraic structures, data structures and algorithms. This book introduces the *require* clause on algorithms in order to achieve constrained genericity.

In order to pursue the introduction of concepts into the C++ language, a workshop was held in 2012 and its summary was published in "Design of Concept Libraries for C++" [104]. It is referred to as the *Palo Alto* report, and it summarizes what design the committee wanted for concepts and what problem(s) it would solve [103]. Indeed, *Elements of Programming* argues that just having constrained template was already incredibly useful, and the STL could be described in terms of *require* clauses (as in requiring a behavior). This subset becomes known as "concepts light" and was enriched to become later what would be standardized in C++20.

Stepanov then published *From mathematics to generic programming* [119] that traces the history of algorithms and ties the history of mathematics with the history of generic programming. Indeed, Stepanov already gave a lecture in 2003 [55] where he traces, in particular, the history of the algorithm of gcd/gcm for 2500 years and explains how successive generation of mathematicians improved it always by looking for more generic ways to solve the same problem. In essence, the book presents generic programming as an extension of the evolution of mathematical algorithms over time. In this book, Stepanov also reclaims the term of *generic programming* and differentiates it from template metaprogramming once and for all. Stepanov also states that "Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth [117] and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream."

Stepanov thought of STL as a starting point to a very large library of data structures and algorithms, written in a generic form, that would all work together nicely. STL is intended to be an example of how industry should move forward and work on building a large library of this form. Hopefully, this is the direction the C++ standard committee is going toward.

Genericity within libraries It is described by the cardinality of how many use-cases it can handle. Libraries always provides their own data structures, to represent and to give a meaning to the data the user wants to process, as well as algorithms to process those data and provide different type of results. A library will be then labeled as *generic* [15] when (i) its data structures allow the user to express himself fully with no limitation and when (ii) its algorithm bank is large enough to do anything the user would want to do with its data. In reality such a library does not exist and there are always limitations. Studying those limitations and what justify them is the key to understand how to surpass them in the future, by developing new hardware and/or software support for new features allowing more genericity.

Genericity within programming languages It is described by the ability of the language to execute the same code over a large amount of data structures [31], be they native (char, int, ...) or user defined. It is nowadays primordial for a programming language to be able to do so. Indeed, in a world where Information Technologies are everywhere, the amount of code written by software developers is staggering. And with it so is the amount of bugs and security vulnerabilities. Being able to natively have a programming language that allows to do *more* by writing *less* mathematically results in a reduced development and maintenance cost. Programming languages offer many ways to achieve genericity which is dependent to the language intrinsic specificities: compiled or interpreted, native or emulated, etc.

Before delving into the specifics of what genericity implies for libraries and programming languages, let us introduce some vocabulary for the sake of comprehension. First is the notion of *type*. A *type* (or *data type*) is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data. Most programming language support basic data

types (also called primitive types) such as integer numbers, floating point numbers, boolean and character strings (ASCII, Unicode, etc.). This data attribute defines the operations that can be performed on the data, the meaning of the data and the size of the data in memory (the data can then be stored on the heap, stack, etc.). A data type provides a set of values from which an expression (i.e. variable, function, etc.) may take those values. Among programming language, we can distinguish those which are dynamically typed and those that are statically typed. Statically typed languages are those whose variables are declared holding a specific type. This variable cannot hold data from another type in the scope it is declared. Statically typed programming languages are Ada, C, C++, Java, Rust, Go, Scala. Dynamically types languages are those whose variables can be reassigned with a value of different type from the one it was initially declared to hold. The variable type is then dynamically changed to fit the new value it is holding. Dynamically typed programming languages are PHP, Python, JavaScript, Perl.

The consequence of being able to tell which type a variable is holding at all time (statically-typed language) is two-fold. For the developer, it is easier to reason about code and to spot bugs. For the compiler, it is possible to generate optimized binary code specific to this data type (vectorization, etc.). The consequence of being able to morph the types a variable can hold at runtime is mainly to serve prototyping purpose. When prototyping in a Jupyter notebook, it is much appreciated not to be limited to a single type for each variable so that we are able to iterate on the prototype much faster.

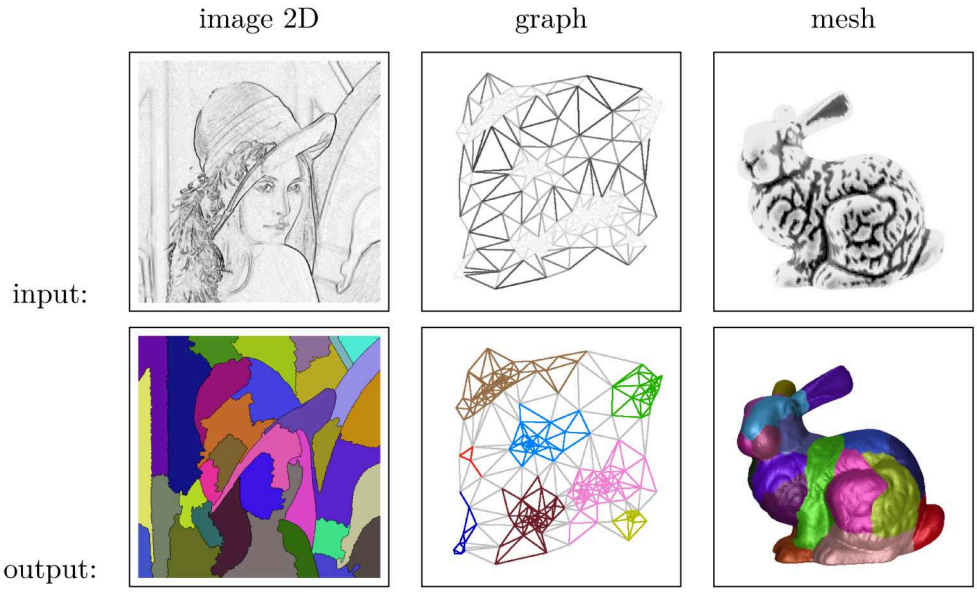
In image processing, an image Im is defined on a domain $\mathcal{D}$ (which contains points) by the relation $\forall x \in \mathcal{D}, y = Im(x)$ where y is the value of the image Im for the point x. This definition always translates into a complex data structure when transposed into a programming language. This data structure must be aware of the data buffer containing the image data as well as information about the size and dimensions of the image. Furthermore, to add to the difficulty, the information needed to define precisely the data structure is not always known when writing the source code. Indeed, a very simple use-case consists in reading an image from a file to load it in memory. The file can contain an image of varying data type and the program should still work properly. There are multiple approach to solve this issue, and we will address them in the following section 2.1 and section 2.2.

2.1 Genericity within libraries

Projecting the notion of genericity to Image Processing, we can deduce that we need two important aspects in order to be generic. First, we need to decorrelate the data structures from its topology, and the underlying data from the algorithms. Indeed, we want our algorithms to support as much data structures as possible. Second, many algorithms share the same computational shape and can be factorized together.

Genericity can have two different meanings depending on the people you ask. For instance, some will argue that genericity is high level and qualifies a tool which is “generic enough” to handle all of his use-cases. Others will argue that genericity is about how a machine (code) is able to make tools, meaning “generic enough” to make a lot of different other tools. Neither is wrong. However, for the sake of comprehension we will use different words for each of these cases. A tool generic enough to handle a lot of use-case will be called *versatile*. Finally, for a tool whose aim is to provide a programming framework to handle the code of any use-case, we will use *generic*. In this thesis, genericity will be about code. The fig. 1 illustrates this result of the same generic watershed implementation applied on a 2D image, a graph and a mesh.

In image processing, there are three main axes around which genericity is revolving. The first axis is about the data type: gray level or RGB color (8-bits, 10-bits), decimal (double) and so on. The second axis, is about the structure of the image: a contiguous buffer (2D or 3D), a graph, a look-up table and so on. Finally, the third axis is about additional data that can be fed to image processing algorithms: structuring element (disc, ball, square, cube), labels (classification),



The same code run on all these inputs.

Figure 2.1: Watershed algorithm applied to three different image types [118].

maps, border information and so on. In the end, an image is just a point within this space of possibilities, illustrated in fig. 2.2. Nowadays, it is not reasonable to have specific code for every existing possibility within this space. It is all the more true when one wants efficiency.

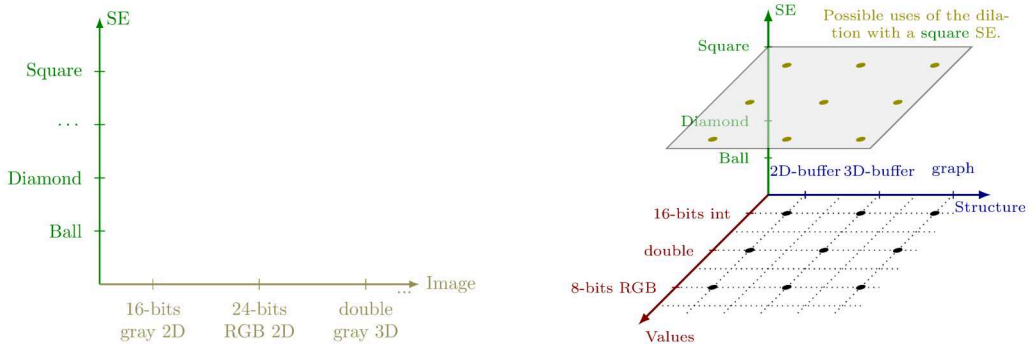


Figure 2.2: The space of possible implementation of the *dilation(image, se)* routine.
The image axis shown in (a) is in-fact multidimensional and should be considered 2D as in (b).

Genericity is not new and was first introduced in 1988 by Musser et al. [8]. The main point is to dissociate data structures and algorithms. The more your data structures and algorithms are tied together, the less you are generic and fail to handle multiple data structures in the same algorithm. Further work has been made about genericity in [15, 31]. Those works highlight the notion of abstraction able to turn an algorithm tied to a data structure into a generic algorithm. Notably in [82], Stepanov digs further and introduce the notion of *Concepts*, which are static requirements about the behavior of a type, by showing how to design a generic library and its algorithms. He highlights the importance of having the algorithms driving the behavior requirements, and not the type. These works are very suitable to be applied in the area of Image processing where we typically have a lot of algorithms (also called operators) that are required to work over a lot of different data structures (also called image types).

The authors explain in [155] how to capitalize on those works to turn a image processing algorithm tied to a data structure into a generic algorithm. We also explain how *concepts* can ease the implementation of generic algorithms. This approach is implemented in a library [140] which allows us to provide a proof of concept over the feasibility of having generic image processing operators running on multiple image types with near-native performance. Let us first explain briefly how we have achieved this.

2.1.1 Different approaches to obtain genericity

First, let us consider the morphological *dilation* that takes two inputs: an image and a flat structuring element (SE). The set of some possible inputs is depicted in fig. 2.2. Without genericity, with s the number of image type, v the number of value type and k the number of structuring elements, one would have to write $s * v * k$ different *dilation* routine.

There are several ways to reach a high level of genericity. First are the *code duplication* and the *generalization* approaches. Finally, there is a way that consists in using expert, domain specific tools specifically engineered for this purpose and build upon them: those tools usually make heavy usage of *inclusion & parametric polymorphism*, also known resp. as object oriented programming and template metaprogramming in C++, to provide the basic bricks to the user.

Code duplication approach It consists in writing and optimizing the algorithm for a particular type in mind. But, each time a new type is introduced, all the algorithms must be rewritten for this specific type. Additionally, each time a new algorithm is introduced, it must support all the existing types and thus be written multiple times. This approach does not scale well when the complexity of algorithms grows, and the number of data types increases. Neither does it allow the implementer to easily make use of optimization opportunities offered by having different data types sharing a property. This often translates into heavy switch/case statement in the code as show in fig. 2.3 that illustrate how the *fill* algorithm needs to dispatch according to the input data type.

```
1  // image types parametrized by their
2  // underlying value type
3  template <ValueType V> struct image2d<V> { /* ... */ };
4  template <ValueType V> struct image_lut<V> {/* ... */};
5  // ...
6  void fill(any_image img, any_value v)
7  {
8    switch((img.structure_kind, img.value_kind))
9    {
10   case (BUFFER2D, UINT8):
11     fill_img2d_uint8( (image2d<uint8>) img,
12                       (uint8) any_value );
13   // ...
14   case (LUT, RGB8):
15     fill_lut_rgb8( (image_lut<rgb8>) img,
16                    (rgb8) any_value );
17   }
18 }
```

Figure 2.3: Fill algorithm skeleton with a switch/case dispatcher to ensure completeness.

In addition, it is important to note that the completeness aspect is only illustrated with regard to the data structure types here. Indeed, the data structures are all already generic for their underlying data type (named *ValueType* in the code). When one write `image2d<uint8>` (l.10), it means *2D-image whose pixels' have a single channel 8-bits value*. This approach enables one to write an algorithm at maximum efficiency for a particular data type, however one can easily miss optimization opportunities if not knowledgeable enough too. This approach is best for early prototypes and trying to find common behaviors pattern among algorithms, or common

properties across different data types. No IP library has chosen this approach due to the obvious maintenance issue induced.

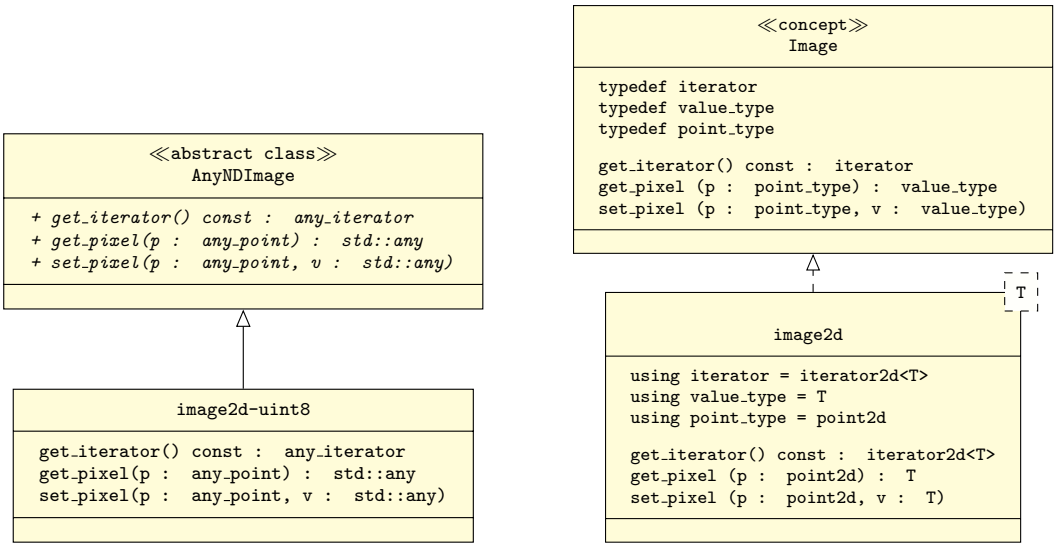
Generalization approach It consists in finding a common denominator to all the image types. Once designed, this common denominator, also called super-type, can store information about all the supported image types by the library. This super-type enables the library developer to write all the algorithms only once: for the super-type. The processing pipeline will then consist in three steps. First convert the input image type into the super-type, second process the super-type into the algorithm pipeline requested by the user, finally convert back the resulting image into the specific image type the user is expecting. This approach offers the advantage of being maintainable. Adding a new image type is just a matter of providing the two conversions facilities: to and from the super-type. Adding an algorithm is also just a matter of writing it once for the super-type. This mechanism is shown in fig. 2.4. However, one must keep in mind that the conversion can be costly. Also, processing the super-type may induce a significant performance trade-off while processing the original type would have been much faster. Furthermore, it is not always possible to find this common denominator when enumerating through some esoteric data types. Finally, the provided interface (from the super-type) may allow the image to be used incorrectly, such as a $2D$ image being processed into video ($3D + t$) algorithm. Widely used libraries such as OpenCV [29] or Scikit-image [120] use this technique to handle as many image types as possible. Another library making use of this generalization technique in its implementation is CImg [105]. CImg generalize its data type to a 4D image type templated by its underlying data type.

```
struct image4D { // generalized super-type
  // generalized underlying value-type
  // every value is converted to this one
  using value_type = std::array<double, 4>;
  /* ... */
 };
// specific types w/ conversion routines
struct image2D { image4D to(); void from(image4D); };
struct image3D { image4D to(); void from(image4D); };
// ...
void fill(image4D img, const std::array<double, 4>& v) {
  for(auto p : img.pixels())
    p.val() = v;
}
```

Figure 2.4: Fill algorithm for a generalized super-type.

Inclusion & Parametric polymorphism approaches They consist in extracting behavior patterns from algorithms to group them into logical brick called *concepts* (for static parametric polymorphism), or *interface* (for dynamic inclusion polymorphism). Each algorithm will require a set of behavior pattern that the inputs need to satisfy. In C++, this technique is achieved either by using inclusion polymorphism, or by using parametric polymorphism [72], as shown in fig. 2.5. In [155], we leverage a new C++20 feature (the concept) to show how it is possible to turn an algorithm, specific to an image type, into a more abstract, generic one that does not induce any performance loss. These approaches, especially applied to image processing, will be seen more in-depth in chapter 3.

Multiple libraries exist and leverage these approaches to try to achieve a high genericity degree as well as high performance by offering varied abstract facilities over image types and underlying data types. Those are ITK [110, 51], Boost.GIL [63], Vigra [37], Higra [153], GrAL [62], DGTal [125], Milena [96, 80, 85], Olena [191, 34, 92, 95, 118] and Pylena [140]. Most of them have been written in complex C++ whose details remain visible from the user standpoint and




```
void fill(AnyNDImage& in,
          std::any val)
{
  for (any_point p :
       in.get_iterator())
    in.set_pixel(p, val);
}
```

(c)

```
template <class InImage>
void fill(InImage& in,
  InImage::value_type val)
{
  for (auto p :
       in.get_iterator())
    in.set_pixel(p, val);
}
```

(d)

Figure 2.5: Dynamic, object-oriented polymorphism (a) vs. static, parametric polymorphism (b).

thus are often difficult and complex to use. It is also harder to debug because errors in highly templated code shows up very deep in compiler error trace.

The table comparing all the pros. and cons. from the aforementioned approaches is presented in table 2.1. We can see in this table that Generic Programming in C++20 check all the boxes that we are interested in.

2.1.2 Unjustified limitations

The image processing community operates mostly with either Python or Matlab [48]. As such this subsection will focus on those two technologies. Python offers access to two major libraries for image processing: OpenCV and Scikit-image. Matlab has built-in support as well as toolboxes for more advanced features. When we intersect Scikit-image and Matlab, we can notice that both are very similar both in terms of feature and interface. As such, it is possible to regroup them both here for the sake of comprehension. As stated above, when considering a generic library, one must consider the three axes: underlying data type, domain structure and additional data. Let us compare how the mentioned library behave along those axes with a simple algorithm such as the morphological dilation.

Limitations regarding feasibility

Data type Dilating a grayscale or a binary image works fine as intended with all the libraries. However, there is no trivial way of dilating an RGB colored image [71, 61] as this operation is not defined for colored images. Indeed, the algorithm is able to work if a supremum function is provided. Such functions have multiple possible implementation and selecting the correct one is not trivial. However, provided a supremum function, the dilation algorithm should normally

Table 2.1: Genericity approaches: .pros & cons.

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.

work. Despite that fact, Scikit-image does not allow to dilate a colored RGB image and raises an error: it is required to convert it into a grayscale the image beforehand.

OpenCV arbitrarily decides that the colored dilation consists in dilating each channel of the colored image separately from one another. It is effectively selecting a partial marginal order relation under the hood. This arbitrary choice may cause false colors to appear in the resulting image (which most of the time is not what the user wants). Furthermore, it is not possible to provide a supremum function to the dilation algorithm to customize the behavior which is a server drawback.

Domain structure To perform a dilation, it is required to have a structuring element whose shape matches the structure of the domain of the image. For instance, dilating a 2D-image requires using a structuring element whose shape may be a disc or a rectangle. To dilate a 3D-image, one would need to use a structuring element whose shape is a ball or a cube. Scikit-image supports 3D-images as well as structuring element whose shapes are compatible (ball, rectangle and octahedron). This naturally leads to having a support for the dilation of 3D-images. On the other hand, OpenCV, as a library, does support 3D-images whereas its dilation algorithm does not. The algorithm exits with an error. Worse, when passing a wrong structuring element (a rectangle) to the dilation algorithm, alongside a 3D-image, the algorithm works and produce a result which is false: it is similar to the application of the 2D-structuring element on each 2D-slice of the 3D-image.

Limitations regarding optimizations

Each library has its own strategies to optimize its routines when implementing them.

Scikit-image It will check whether the structuring element is separable (only for rectangle shapes) so that it can dispatch on an optimized multi-pass 1D routine for each part separated, which linearizes the execution time and greatly improve performance for large structuring element.

Also, Scikit-image relies on SciPy internals which does not abstract the underlying data type for the algorithm implementer. As such, each algorithm must provide a switch/case dispatch for every supported type (floating points, 8-bits channel, 16-bits channel, RGB, etc.), and it must provide it in the middle of the algorithm implementation. If one type is not natively supported, an error occurs and the program halts. Henceforth, handling a new supported data type will requires to review every single already written algorithm.

On the other hand, SciPy provides an abstraction layer over the dimensional aspect of the image by providing a tool named point iterator. This tool allows one to iterate over every point of the image, without being aware of the number of its dimension, and make the translation from the abstract iterator to the actual offset in the data buffer of the image. The implementer can then only worry about handling the underlying data type to provide a generic algorithm. This approach, sadly, is fully dynamic (that is, runtime) and does not allow the compiler to provide native optimization such as vectorization out of the box.

OpenCV & Matlab In OpenCV as well as in Matlab, the choice was made to systematically attempt to decompose big rectangular structuring elements into smaller $3 * 3$ structuring elements. This is not as effective as using multi-pass 1D algorithm but still allows for relatively stable performance.

Also, OpenCV let the implementer handle the cases he wants to support by himself. For instance, the dilation algorithm is written with a dispatch on the data type before the actual call to the algorithm. This enables compiler optimizations such as vectorization because all the required information is known at the right time. It also enables offloading the computation into GPU kernels when feasible. However, the downside is that few algorithms are written in a way to handle multidimensional images. Most are written to only handle specific subsets. As such, conversion from one subset to another may be unavoidable when writing an algorithm pipeline for a more complex application. For instance, it is currently not possible to dilate a 3D image with a 3D ball (as stated above).

Another point to note with OpenCV is the requirement to do temporary copies (to extract data or to have working copy for in-place computation) when writing an algorithm. For instance, it is currently not possible to write a dilation algorithm operating only on the green channel of an RGB image. One must first extract the green channel into a single channel temporary image, blur that image, to finally put the result back into the original image. Generally, in-place computation is poorly handled in OpenCV.

Performance discussion When comparing performance of the simple dilation between Matlab and OpenCV, which is done in [102], shows that Matlab is very oriented toward prototyping and not toward production. The performance gap between the two libraries shows that performance may not a major concern for MATLAB in this case. Opposite to this, OpenCV and Scikit-image both have a C/C++ core to provide fast basic algorithms such as the dilation and erosion mathematical morphology.

As such, when comparing the performance of OpenCV, Scikit-image and Pylene in fig. 2.6, we can notice some interesting facts. Both Scikit-image and Pylene have a very stable execution time even though the size of the structuring element grows by power of two. This corroborates the fact that the author did see code taking advantage of the structuring element's properties, such as the decomposability/separability. OpenCV has very good performance for a square because it has specific handwritten code for both vectorization and GPU offloading when possible: even if OpenCV decomposed its square into smaller sub square (and not periodic lines), it remains steady fast.

In the case of a structuring element shaped as a disc (also in fig. 2.6), we can observe that the execution time raises exponentially for both Scikit-image and OpenCV whereas Pylene remains regular and steady fast. These results show that Pylene's attempt to decompose each structuring element into periodic lines when possible may be slightly slower for smaller structuring elements whereas it is much more stable and faster when the structuring element start to be of a certain size.

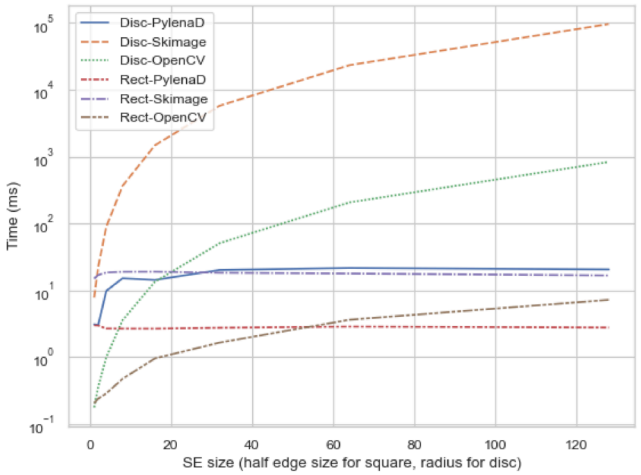


Figure 2.6: Benchmark: dilation of a 2D image ($3128 \times 3128 \approx$10Mpix) with a 2D square and a 2D disc.

Limitations regarding static types in a dynamic world

Being statically typed (at language level) has its advantages (performance, optimization) but also has sever drawbacks, especially when interfacing with dynamic languages such as Python. Indeed, being statically typed in a dynamic world disqualify the library from being able to select, for instance, a new, custom structuring element, when performing a dilation. As a matter of fact, all the supported structuring elements must be listed in advance and code must be compiled for each supported type. This induces a strong inertia when the practitioner wants to try out new things not yet supported by the library. This also defeats the initial purpose of being generic. Indeed, one would expect a generic library to work with any type and not just a specified set of pre-compiled types. Being able to break through this limit is addressed more in-depth in chapter 5.

2.1.3 Summary

To conclude, a generic algorithm will not be faster after it is first written, but will provide acceptable performance for most cases. However, a generic algorithm provide opportunities for the implementer to take advantage of some properties from input types in order to be faster. The main advantage is that once those optimization opportunities are written once, they are available for every input types (that match the property). Genericity enables code dispatching very easily based on type properties which may induce almost no runtime overhead, depending on which generic strategy was chosen in the library.

2.2 Genericity within programming languages

Genericity is a more than 45 years old notion. It was first introduced alongside the CLU language in 1974 by Barbara Liskov and her students [14]. The language offered many features such as data encapsulation, iterators and especially *parametrized modules*. A module in CLU is represented by a *cluster*. A *cluster* is a programming unit grouping a data structure and its related operations. In modern programming, we would call it a class where the data structure s to the member variables and the operations correspond to the member functions (or methods). In CLU, the clusters can be parametrized which is a way to introduce the notion of parametric polymorphism. Indeed, clusters offer the ability to define a generic data structure and its functions whose behavior will

not change whatever type the cluster is parametrized with. At first, type-safety was enforced at runtime in CLU. Later, *where clauses* were introduced to specify specific requirements over cluster type parameters. This had the consequence of allowing only the operations required in the where clause to be used in the cluster implementation code. This enables proper compile-type checks, type-safety and the compilation into a simple native and optimized code by the compiler. The following CLU code illustrates how to create a cluster (or class) named *vector*, declaring a set of member functions and requiring a specific behavior (operator = and <) from its underlying stored elements:

```
vector = cluster [T: type] is
  create, size, contains, sort, remove, push_back
where
  T has equal: prototype(T, T) return (bool)
  T has less: prototype(T, T) return (bool)
```

When implementing the member functions declared for `vector`, the only valid operations that can be performed on a value of type `T` are `equal` (= for comparison) and `less` (< for ordering). In CLU the actual instantiation of the parameter (here `T`) is done at runtime. Indeed, each type is represented by a descriptor, even a parametrized type parameter. At runtime, a concrete type is used to instantiate the cluster. To achieve this, the placeholder type descriptor is replaced by the one of the actual concrete type at this moment. The instantiation can then be considered dynamic which differs from the C++ compilation model where the template instantiation is considered static (i.e. done at compile-time). The pros. and cons. of this approach are discussed in [1] which essentially summarize to the resulting (static) binary code size exploding due to combinatorial explosion combined with optimized code generation versus small, flexible and slower binary (dynamic instantiation).

Later on came the Ada programming language whose conception work started in 1977 and whose first version was released in 1980. It was first standardized in 1983 [3] in the United States by the American National Standards Institute (ANSI) before being Internationally standardized by the International Organization for Standardization (ISO) in 1987 [6]. From this point on, the Ada language released a new version in 1995 [18, 43], then the standard committee published an Amendment in 2007 [73] which is often referred as Ada 2005. Finally, a new version of the language was published in 2012 [100, 128]. What interests us in the Ada language is the fact that the language features *generics* since it was first designed in 1977–1980, which is *ten years* before Musser and Stepanov published their first work about genericity in 1988 [8]. Also, it took ten more years for the fist Standard Template Library (STL) to be standardized in the C++ programming language in C++98 [25] afterwards. Finally, it took almost ten more years for an STL to land into the Ada programming language in the 2007 Amendment for the Ada programming language standard (also named Ada 2005).

In the Ada programming language, it is possible to mark a package or a procedure as generic with the `generic` keyword. The developer then lists the parametrized parameter the package and/or the function requires to be implemented. The following code demonstrates how this feature works by implementing a function replacing the first argument by a new value and returning its previous value (also known as exchange).

```
generic
  type T is private;
function exchange (x : in out T; v : in T) return T is
  tmp : T;
begin
  tmp := x;
  x   := v;
  return tmp;
end exchange;
```

In Ada, generic packages and routines (procedures, functions) must be instantiated explicitly: the compiler cannot infer the parametrized type at compile-time from the context of use (whereas

a C++ compiler can). Thus, the following three lines must be written explicitly for the compiler to generate the binary code of the function above:

```
function int_exchange is new exchange (Integer);
function float_exchange is new exchange (Float);
function str_exchange is new exchange (String);
```

In Ada, this model enables the possibility of sharing a generic across several compilation units since its compilation is independent of its use whereas in C++, until C++20 the sharing model consists in copy-pasting the whole source code (and transitive recursive dependencies) each time a generic (a template) code is compiled. C++20 (standardized in 2020, 22 years after C++98) brings a solution to this issue by standardizing C++ modules. However, this feature is out of scope of this study and will not be discussed in this thesis.

Also, Ada support syntactics constraints on parameters similar to the CLU language. It is translated into a `with` clause listing the constraints on the parameter(s). The following code shows how to implement such a constrained generic function by using the mathematical maximum operator as an example:

```
generic
  type T is private;
  with function "<" (x, y: T) return Boolean is <>;
function maximum (x, y : in T) return T is
  tmp : T;
begin
  if x < y then
    return x;
  else
    return y;
  end if;
end maximum;
```

Let us not forget to instantiate the constrained generic function for integers:

```
function int_maximum is new maximum (Integer);
```

This idea of constrained generic is 45 years old (the genesis was in CLU) and has only made his way very recently (2020) in the C++ programming language under the feature named *concepts*.

Now that we have introduced the origin of generic programming, let us focus on the C++ programming language and how generic programming has evolved in C++ for the last 30 years, and how it will evolve in the future.

2.2.1 Genericity in pre C++11

Before C++11 [90] came out in 2011, the genericity facilities offered by the C++ programming language were already Turing-complete [56]. However, it was lacking some key features the language now have, which made writing generic code a real challenge at that time. For instance, in c++98 [25] (released in 1998), when writing code with a variable number of type parameters (nowadays designated as *variadic templates*) one had to write the generic code for each and every number of type parameter supported. This meant that to implement `std::tuple`, one had to copy the implementation for every number of type supported by `std::tuple`. This limitation defeated the very first principle and motivation of generic programming which is to write *less* code. To compensate, library implementers used tricks with macros not to have to rewrite code, which made the initial code even harder to understand for outsiders.

Functions on types (or metafunctions, or traits)

The very first feature every developer writing metaprogramming C++ code has used is called *type-traits*. Those *traits* are a way to mutate a type depending on the way a template declared

on a data structure is resolved. Indeed, in C++ the instantiation of a template depends on the context, which means the compiler is required to build a set of possible way to resolve a template instantiation in order to determine the best match to resolve this instantiation. This mechanism is well known and documented in the C++ standard, it can then be used (and abused) to do a great number of things. Among all the traits that exist (a lot of them have been standardized in C++11, 2011, in the header `<type_traits>`), some in particular are used in every codebase: `remove_const`, `remove_volatile` and `remove_reference`. Let us see how they are implemented:

```
template<class T> struct remove_const                { using type = T; };
template<class T> struct remove_const<const T>       { using type = T; };

template<class T> struct remove_volatile             { using type = T; };
template<class T> struct remove_volatile<volatile T> { using type = T; };

template<class T> struct remove_reference            { using type = T; };
template<class T> struct remove_reference<T &>       { using type = T; };
template<class T> struct remove_reference<T &&>      { using type = T; };
```

For `remove_const`, first is defined the structure whose underlying alias `type` points to the passed template parameter `T`. Then we define a template specialization whose matching parameters are all `T` parameters that are `const`. The defined underlying alias `type` for this specialization then is `T` without the qualifying `const`. This way, there are two possibilities when calling this `trait`: either the passed type parameter is not `const` which means it will be forwarded as-is to the underlying alias `type`, or the passed type parameter is `const` which means the underlying alias `type` will be defined by dropping the `const-qualifier` off of the passed parameter type. For instance:

```
using T1 = remove_const<double>::type // T1 is double
using T2 = remove_const<const double>::type // T2 is double too, const-qualifier is dropped
```

This language construct is very useful when developing generic libraries because it allows performing "functions" on types, and even chaining them. It is also possible to perform checks to extract information about those types. We can easily write an `is_const` metafunction if we need it.

In image processing in particular, the usage of traits in the generic library Milena was very useful to achieve standard ways to compute very useful types from other complex type. From the image processing definition of an image type, we can already see a number of traits that a generic library wants to provide. Indeed, it is useful to be able to extract the type of the domain (box2d, box3d, etc.), the type of the point (point2d, point3d, ...), the underlying value type (uint8, rgb8, etc.) and so on. We can also already see emerging consistency issues between those types. Indeed, a box3d domain should not accept access via a point2D. It shall instead require a point3d. All those issues will be addressed later in the chapter 3. However, we can already give here minimal working example as to how type traits are useful in a generic image processing library. First let us define some minimal data structures:

```
struct point2d {int x, y; };
struct rgb8 { uint8_t r, g, b; };
struct box2d {
  using value_type = point2D;
  // ...
};
struct image2d {
  using domain_type = box2d;
  using point_type  = point2d;
  using value_type  = rgb8;
  // ...
};
```

Now we want to implement `traits` to extract type information from those structures. Here is how we do it in a generic library:

```
template <class T> struct domain_value_type { using type = typename T::value_type;  };
template <class T> struct image_point_type  { using type = typename T::point_type;  };
template <class T> struct image_value_type  { using type = typename T::value_type;  };
template <class T> struct image_domain_type { using type = typename T::domain_type; };
```

These traits extract information about types in a generic way and can be used in any algorithm taking an image as a template parameter. For instance, here is how an image processing algorithm (trivially extracting the max value) would be written:

```
template <class Ima>
typename image_value_type<Ima>::type // usage of trait
min_value(const Ima& ima) {
  using value_t = typename image_value_type<Ima>::type; // usage of trait
  auto min = std::numeric_limits<value_t>::max();
  for(auto v : ima.values()) {
    auto min = std::min(min, v);
  }
  return min;
}
```

SFINAE: Substitution-Failure-Is-Not-An-Error

Another feature related to metaprogramming allowed the developer to design generic libraries: the SFINAE [49] (substitution-failure-is-not-an-error) technique that leads to the popularization of the usage of the `std::enable_if` metaprogramming construct. The SFINAE technique relies on a feature of the C++ programming language. Indeed, when standardizing how the compiler should resolve and select function overloads, in a templated context, the standard committee chose to have the following behavior: "when substituting the explicitly specified or deduced type for the template parameter fails, the specialization (function overload candidate) is discarded from the overload set (of matching functions) instead of causing an error".

This feature allows writing code that seems to be ill-formed, for instance in a function, trying to access a class member type, variable or function that does not exist. However, because it happens in a templated context during the instantiation resolution, when the compiler tries to instantiate a function template with a parametrized type, the compiler will just discard the function from the overload resolution set at call-site instead of halting with a hard error. An error can occur only when the compiler tried all the overload it knows in the overload set and still could not find an overload that was not ill-formed. If this happens, the compiler will then proceed to list all the overloads it tried, to list all the template substitution combinations it tried and finally to list why it failed. This mechanism is the very reason of the unpopularity of this technique as it leads to situation where the compiler can output *several Mos* of error message for one single source file. Error messages become incomprehensible very fast and programs are hard to debug because everything happens at compile-time. But still, it was the only technique available to perform any kind of detections on types at compile-time, or to require any behavioral constraints on them.

For instance, here is some real-world example extracted from generic image processing code that allows implementing the *fill* algorithm in two very different ways depending on how behaves the input image type. First we need to write the detector which is a structure whose templated context will be ill-formed during template instantiation. This detector will inherit either `std::true_type` or `false_type` depending on whether the detection is successful or not:

```
// Step 1 write detector
template <class Ima, class = void>
struct is_image_with_lut : std::false_type {};

template <class Ima>
struct is_image_with_lut<Ima,
   typename Ima::lut_type // constraint over the existence of the lut_type field
> : std::true_type {};
```

Now let us introduce a new image type for the sake of this example:

```
struct image2d_lut : image2d {
  using lut_type = std::array<uint8_t, 256>;
  using value_type = uint8_t:
  lut_type& get_lut();
};
template <class Ima>
struct image_lut_type {
  using type = typename Ima::lut_type;
}
```

The next step is to implement the `enable_if` facility. This is included in the C++ STL starting from C++11 and onward:

```
template<bool B, class = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> {
  using type = T;
};
```

Now we are all set to use all those constructs to dispatch our algorithms from the call-site depending on our input image type:

```
// Overload #1 : with lut
template <class Ima,
void fill(Ima& ima, typename image_value_type<Ima>::type val,
          typename enable_if<is_image_with_lut<Ima>::type::value, void*>::type = 0)
{ // Image with lut
  using lut_t = typename image_lut_type<Ima>::type;
  using value_t = typename image_value_type<Ima>::type;
  lut_t& lut = ima.get_lut();
  for(value_t& v : lut)
    v = val;
}

// Overload #2 : without lut
template <class Ima,
void fill(Ima& ima, typename image_value_type<Ima>::type val,
          typename enable_if<not is_image_with_lut<Ima>::type::value, void*>::type = 0)
{ // Image without lut
  using value_t = typename image_value_type<Ima>::type;
  for(value_t& v : ima.values())
    v = val;
}
```

Finally, let us give some call-site example to finish illustrating our point:

```
image2d_lut ima1;
image2d ima2;

fill(ima1, 0);   // will dispatch over overload #1
fill(ima2, 255); // will dispatch over overload #2
```

Here, no hard error occurs. Both overloads are dispatched according to the constraint built with the SFINAE construct. Now we can talk about *constrained genericity* in C++. If we had an algorithm returning a value instead of performing an in-place computation, we would have been able to write the code with the SFINAE construct in the return-type instead of having it in the function parameters. However, that is not the case here.

CRTP: Curiously Recurring Template Pattern

Another feature that precedes C++11 and was available in C++98 [25] and C++03 [54] is the curiously recurring template pattern (CRTP) introduced in 1996 by Coplien [22]. This

programming technique allows a base class (in its specific code) to be aware of its derived class at compile type. We made extensive use of this pattern in the past to design a new paradigm, SCOOP [53, 65, 78, 92] which combined multiple inheritance (via CRTP) and concept checking (via Boost.Concept [39, 70]) to implement a solution to provide a library with constrained genericity in C++ for the image processing area. The Scoop paradigm relied on the fact that multiple (especially diamond) inheritance did not pose much of an issue as long as there was no member variable involved. This way, one could consider a class hierarchy as a hierarchy of constraints indstead. Using CRTP, it was possible to find back, inside constraining classes, what was the concrete leaf class of the hierarchy in order to check whether its implementation satisfied the constraints or not. For instance, let us assume that we have a concrete class `image2d` inheriting from a constraining class `Image`. Let us now see how one would use the SCOOP paradigm to implement it. First we need to implement the satellite constraints around the image type, which are related to the underlying point and domain of the image.

```
template <class P>
struct Point { // concept checking class
  int (P::*m_ptr1) const = & P::dim;
};

struct point2d : Point<point2d> { // CRTP
  int x, y;
  int dim() const { return 2; }
};

template <class D>
struct Domain { // concept checking class
  using value_type = typename D::value_type;
  // ...
};

struct box2d : Domain<box2d> { // CRTP
  using value_type = point2d;
  // ...
};
```

For the sake of brevity, we are omitting the implementation of domain 's member functions related to iterating over all the points of the domain. With those concepts defined, we can now dig into how we would implement an image class with the SCOOP paradigm.

```
template <class I>
struct Image { // concept checking class
  using value_type = typename I::value_type;
  using point_type = Point<typename I::point_type>;
  using domain_type = Domain<typename I::domain_type>;

  const domain_type& (I::*m_ptr1)() const = & I::domain;
  // ...
};

struct image2d : Image<image2d> { // CRTP
  using value_type = uint8_t;
  using point_type = point2d;
  using domain_type = box2d;

  // ... ctors ...

  const domain_type& domain() const {
    return m_dom;
  }
private:
  domain_type m_dom;
};
```

Each time a leaf class in the hierarchy inherits from a base class, the concepts are checked. The syntax is not really intuitive, especially writing the function pointers to check that their

prototype corresponds to certain behavior, but it was all that was available at that time. To be completely exhaustive, the library implementing this paradigm, Milena, featured another powerful tool which enabled one to require only concept class into input data for algorithms. Inside the algorithm it was then possible to get back the concrete class and use it as if it was originally passed as an argument. To do so, it was needed to add a member field to the concept class named `exact_t` that kept track of the leaf class into the concept checking class.

```
template <class I>
class Image {
  // ...
  using exact_t = I;
};
```

Then a simple cast routine would do the trick inside the algorithm:

```
#define EXACT(Ima) \
  typename Ima::exact_t

template <class Ima> // mutable routine
EXACT(Ima)& exact(Ima& ima) { return static_cast<EXACT(Ima)&>(ima); }

template <class Ima> // const routine
const EXACT(Ima)& exact(const Ima& ima) { return static_cast<const EXACT(Ima)&>(ima); }
```

This way, an image processing algorithms, the implementer would only need to write minimal code for it to work out of the box:

```
template <class I>
void fill(Image<I>& ima, typename image_value_type<Ima>::type val) {
  EXACT(Ima)& ima_ = exact(ima);

  // use the concrete underlying image ima_ and val
}
```

This constrained genericity would be totally transparent as shown with the following code that just works out of the box, thanks to the SCOOP paradigm and its inheritance strategy to constrain classes.

```
image2d ima = /* ... */;
uint8_t val = 0;
fill(ima, val);
```

By extension, this work on Milena was integrated in the image processing platform Olena [191, 95]. This platform centralizes the work that was done around this field of research for a long time [35, 33, 47, 57]. More details can be found about SCOOP, and notably how it enables property based programming (augmentation of types via properties) in the work of Levillain [92, 80, 85, 86, 93, 92, 101, 118].

Those approaches have the advantage of being really flexible and to be able to perform the constrained genericity that we wanted to have, be it to constrain an implementation or to dispatch to the correct overloaded algorithms depending on specific properties. The disadvantages come in the form of an increased complexity of the design hierarchy of implemented types as they must inherit concepts via CRTP and conform to specific constrains. Also, all the implementation is visible as all the code is generic (template) and thus all the implementation details are leaked to the user code. For an image processing library which can use several dependencies (for instance, a library that read images from disc from multiple image formats), this is a huge drawback.

With the release of new C++ standards in 2011 [90], then 2014 [116] and 2017 [136] where template metaprogramming facilities were greatly improved, it was necessary to review once more this design in order to improve it to achieve genericity, performance and ease of use. This is the birth of a new library, Pylene [140]. In the end, it was C++20 [90] that marked the shift wanted by Stepanov [8, 15, 31, 82] and Stroustrup [16, 27, 185, 76] for years with the arrival of concepts, and all its new possibilities brought to the programmer.

2.2.2 Genericity in post C++11 (C++20 and Concepts)

```
1  template <Image Ima, ValueType Val>
2    requires same<Ima::value_type, V>
3  void fill(Ima ima, Val val) {
4    for(Val v : ima)
5      v = val;
6  }
```

Figure 2.7: Fill algorithm, generic implementation.

Most of the algorithms are *generic* by nature. What limits their genericity is the way they are implemented. This statement is justified by the work achieved in the Standard Template Library (STL) [31] in C++, whose algorithms are implemented and designed in a way where they work with all the built-in collections (linked list, vector, etc.). Let us take the example of the algorithm `fill(Collection c, Value v)` which set the same value for all the element of a collection (see fig. 2.7). There are three main requirements here that are not related to the underlying type of *Collection*. First, we check (l.2 fig. 2.7) that we are actually filling the collection with the correct type of value. Indeed, it would not make sense, for instance, to assign an RGB triplet color into a pixel from a grayscale image. Secondly, we need to be able to iterate over all the of collection's elements (l.4 fig. 2.7). Finally, we need to be able to write a value into the collection (l.5 fig. 2.7). This requires the collection not to be read-only, or the collection's values not to be yielded on-the-fly. This allows us to deduce what is called a *concept*: a breakdown of all the requirement about the behavior of our collection. When writing down what a *concept* should require, one should always respect this rule: "*It is not the types that define the concepts: it is the algorithms*". Concepts in C++ are not new and there have been a long work to introduce them that goes back from 2003 [186, 185, 190] to finally appear in the 2020 standard [192] (referred as C++20 [90]). This allows us, as of today, to write code leveraging this facility.

Conceptification

C++ is a multi-paradigm language that enables the developer to write code that can be *object oriented*, *procedural*, *functional* and *generic*. However, there were limitations that were mostly due to the backward compatibility constraint as well as the zero-cost abstraction principle. In particular the *generic programming* paradigm is provided by the *template metaprogramming* machinery which can be rather obscure and error-prone. Furthermore, when the code is incorrect, due to the nature of templates (and the way they are specified) it is extremely difficult for a compiler to provide a clear and useful error message. To solve this issue, a new facility named *concepts* was brought to the language. It enables the developer to constraint types: we say that the type *models* the *concept(s)*. For instance, to compare two images, a function *compare* would restrict its input image types to the ones whose value type provides the *comparison operator* ==. In spite of the history behind the *concept checking* facilities being very turbulent [186, 185, 190], it finally landed in the last standard [192] (C++20).

The C++ *Standard Template Library* (STL) is a collection of algorithms and data structures that allow the developer to code with generic facilities. For instance, there is a standard way to *reduce* a collection of elements: `std::accumulate` that is agnostic to the underlying collection type. The collection just needs to provide a facility so that it can work. This facility is called *iterator*. All STL algorithms behave this way: the type is a template parameter, so it can be anything. What is important is how this type behaves. Some collection requires you to define a `hash` functions (`std::map`), some requires you to set an *order* on your elements (`std::set`) etc. This emphasis the power of genericity. The most important point to remember here (and explained very early in 1988 [8]) is the answer to: "*What is a generic algorithm?*". The answer is: "*An algorithm is generic when it is expressed in the most abstract way possible*". Later, in his

book [82], Stepanov explained the design decision behind those algorithms as well as an important notion born in the early 2000s: the concepts. The most important point about concepts is that it constrains the behavior. Henceforth: "*It is not the types that define the concepts: it is the algorithms*". The *Image Processing* and *Computer Vision* fields are facing this issue because there are a lot of algorithms, a lot of different kinds of images and a lot of different kinds of requirements/properties for those algorithms to work. In fact, when analyzing the algorithms, you can always extract those requirements in the form of one or several *concepts*. This section is a preface to the image taxonomy which will be seen more in-depth in chapter 3.

Image processing algorithms, similarly, are *generic* by nature [10, 35, 47, 85, 118]. When writing an image processing algorithm, there is always a way to express it with a high level of abstraction. For instance, it is possible to write a morphological dilation in a way that does not care about the underlying value type, the domain nor the structuring element specificities. The most abstract way to write a dilation is shown in fig. 2.8.

```
1  template <Image I, WritableImage O,
2              StructuringElement SE>
3  void dilation(I input, O output, SE se) {
4    assert(input.domain() == output.domain());
5    for(auto pnt : input.points()) {
6      output(p) = input(p)
7      for (nx : se(p))
8        output(p) = max(input(nx), output(p))
9    }
10 }
```

Figure 2.8: Dilation algorithm, generic implementation.

This implementation introduces three concepts at line 1: *Image*, *WritableImage* and *StructuringElement*. Following the behavior of each one of them into the algorithm, we can deduce a list of requirements for each one of them.

Image It is the most basic representation of what an image should be. An image should (a) provide a way to access its domain (l.3 fig. 2.8) and (b) a way to iterate over its points (l.4 fig. 2.8). This then enables us later to (c) access the value returned by the image at this point (l.5 fig. 2.8). Up to this point, the value is only accessed in read-only. We can write the following two concepts:

```
template <typename I>
concept Image = requires {
  typename I::point_range;            // needed for a
  typename I::point_type;             // needed for b
  typename I::value_type;             // needed for b
} && ForwardRange<I::point_range>     // needed for a
&& requires (I ima, I::point_type pnt) {
  { ima.points() } -> I::point_range; // a
  { ima(pnt) }     -> I::value_type;  // b
};
```

In reality, more boilerplate code is needed to ensure, for instance, that there is no type mismatch between the image's `point_type` and the `point_range`'s value type. For the sake of brevity this boilerplate code is omitted here.

WritableImage It is a more specific concept based on the previous *Image* concept. It requires that the image's value can be (d) accessed to be modified: the user should be able to write into the image's value accessed by a specific point (l.6 fig. 2.8). We can then write the following two concepts:

```
template <typename WI>
concept WritableImage = Image<WI>
```

```
&& requires (WI wima, I::point_type pnt,
             I::value_type val) {
  { wima(pnt) = val };                // d
};
```

StructuringElement It is an additional input to the image defining the window around each point that will be considered during the dilation (also called the neighborhood). A structuring element should just provide a list of point when it is input with one (e). From this behavior we can deduce the following concept:

```
template <typename SE, typename I>
concept StructuringElement = Image<I>
&& requires (SE se,I::point_type pnt) {
  { se(pnt) } -> I::point_range;      // e
}
```

This new notion of concept is very important because it decorrelates the requirements on behavior required inside algorithms from the way the data structures are designed. This is a way to always wrap a specific data structure so that it can behave properly into an algorithm, without needing to rewrite that algorithm.

Simplifying code

The main advantage brought by using modern C++ as the implementation language for an image processing library is the possibility to leverage what is called metaprogramming. Metaprogramming is a way to tell the compiler to make decision about which type and which code it generates. These decisions are made at compile-time and are then absent from the resulting binary: only the fast and optimized code remains. This brings a new distinction between the static world (what is decided at compile-time) and the dynamic world (what is decided at runtime). The more is decided at compile-time the smaller, faster the binary will be because there is less work to do at runtime. By following this principle, one can think of some properties that are known ahead of time (at compilation) when writing an image processing algorithm. For instance, when considering the example of the dilation whose code is shown in fig. 2.8, we can see that the decomposability property of the structuring element is linked to the type. This means that when the structuring element's type is a disc, or a square, the compiler will know at compile-time that it is decomposable. To tell the compiler to take advantage of a property at compile-time, C++ has a language construct named *if-constexpr*. The resulting code then becomes:

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

There are other ways to achieve the same result with different language constructs in C++. There are two "legacy" language constructs which are tag dispatching (or overload) and SFINAE. With the release of C++17 came a new language construct presented above: *if-constexpr*. Finally, with C++20, it will be possible to use concepts to achieve the same result. In comparison, to achieve the same result as above with tag dispatching, the following code is needed:

```
struct SE_decomp {};
struct SE_no_decomp {};
```

```
template <Image Img, StructuringElement SE>
auto dilate(Img img, SE se) {
  // either SE_decompo or SE_no_decomp
  return dilate_(img, se, typename SE::decomposable());
}

auto dilate_(Img img, SE se, SE_decomp) {
  lst_small_se = se.decompose();
  for (auto small_se : lst_small_se)
    // Recursive call
    img = dilate(img, small_se, SE_no_decomp)
  return img;
}
auto dilate_(Img img, SE se, SE_no_decomp) {
  if (is_pediodic_line(se))
    return fast_dilate1d(img, se) // Van Herk's algorithm;
  else
    return dilate_normal(img, se) // Classic algorithm;
}
```

To achieve the same result with SFINAE, one would need to write the following code:

```
// SFINAE helper
template <typename SE, typename = void>
struct is_decomposable : std::false_type {};
template <typename SE>
struct is_decomposable<SE,
  // Check wether the type provides the decompose() method
  std::void_t<decltype(std::declval<SE>().decompose())>
> : std::true_type {};
template <typename SE>
constexpr bool is_decomposable_v =
                is_decomposable<SE>::value;

template <Image Img, StructuringElement SE,
  typename = std::enable_if_t<is_decomposable_v<SE>>>
auto dilate(Img img, SE se) {
  lst_small_se = se.decompose();
  for (auto small_se : lst_small_se)
    img = dilate(img, small_se) // Recursive call
  return img;
}

template <Image Img, StructuringElement SE,
  typename = std::enable_if_t<not is_decomposable_v<SE>>>
auto dilate(Img img, SE se) {
  if (is_pediodic_line(se))
    return fast_dilate1d(img, se) // Van Herk's algorithm;
  else
    return dilate_normal(img, se) // Classic algorithm;
}
```

Comparing those two last ways of writing static code to the first one results in an obvious conclusion: the *if-constexpr* facility is much more readable and maintainable than the two legacy ways of doing it. Finally, there is still another way to handle the issue, and it is with C++20's concepts. The following code demonstrates how to leverage this language construct:

```
template <typename SE>
concept SE_decomposable = requires (SE se) {
  se.decompose(); // this method must exist
};

template <typename Img, typename SE>
auto dilate(Img img, SE se) {
 if (is_pediodic_line(se))
    return fast_dilate1d(img, se) // Van Herk's algorithm;
  else
    return dilate_normal(img, se) // Classic algorithm;
}

template <typename Img, typename SE>
```

```
    requires SE_decomposable<SE>
  auto dilate(Img img, SE se) {
    lst_small_se = se.decompose();
    for (auto small_se : lst_small_se)
      img = dilate(img, small_se) // Recursive call
    return img;
  }
```

A best-match mechanic [64] operates under the hood to select the function overload whose concept is the most specialized when possible. The best-match is very interesting to us as it removes completely the need to have mutually exclusive conditions which were required with the SFINAE technique and could result in explosive complexity with the growing number of *and*/*or* clauses in constraints. The best-match mechanic works on top of another widely used machinery in C++: the function overload set. For a specific function call, all the function overloads will be considered in the overload set. Then, to select the correct one, they will be sorted according to the best-match procedure. Finally, if there is still two overloads that have the same priority, the compiler will raise a hard error stating that there is an ambiguity it cannot solve on its own. As a matter of fact, adding one more function to the overload set is enough to have it selected when it matches without modifying the already existing functions. We could have added, in our example, an overload for structuring element whose shape is a periodic line, assuming that this is a property we can detect at compile-time.

2.3 C++ templates in a dynamic world

There are three main categories of programming languages. First are the *compiled* programming languages which require to feed the source code to a program (a compiler) that will output a binary. This binary will then produce the desired output once the user execute it. Some well known languages of this category are C, C++, Ada, Fortran. Second there are the interpreted programming languages which require to feed the source code to a program (an interpreter) that will directly produce the output as if a binary was executed. Some well known languages of this category are Javascript, Python, Matlab, Common Lisp. There is also a third, hybrid category that tries to combine the best of both world. This last one is discussed more in-depth in section 5.1.1. Both categories have advantages and drawbacks.

Compiled languages They are still widely spread and used as of today. They present a working pipeline which is very classic. First the programmer will write code, then the compiler will build a binary optimized for the target architecture and finally the programmer can execute his binary to produce a result. Usually the compilation step is slow whereas executing the binary is fast. There is no additional step when it comes to the binary execution. This, however, has the effect of having a poor portability. Indeed, my binary optimized to use fast and recent SIMD AVX-512 instructions will not work on an old x86 machine that does not support those CPU instructions. When distributing our program, multiple binaries must be produced for each supported CPU architectures. Furthermore, usually compiled languages have very poor support of dynamic language features such as reflection, code evaluation or dynamic typing. It tends to improve with time, but solutions are limited to compile-time information or need to ship a JIT-compiler into the final binary (such as cling [106] for C++) to generate new binary on-the-fly to be executed right after. This has two drawbacks: slowness when compilation is needed and increase of the binary size.

Interpreted languages They are also widely used, especially in the research area where a fast feedback loop between prototyping and getting results is needed. The compilation time is very fast and allows a program to be almost instantly executed. In fact, the compilation can be done just ahead of program execution not to compile unnecessary code. However, the execution time

will generally be slow. To the user it is invisible because both compilation step and execution step are blurred together. Furthermore, most of the time a same interpreted program is executed once. Then the programmer will modify it and continue its prototyping process. The real advantage of an interpreted language is the portability. As it is the responsibility of the client to install the correct language interpreter (for the correct version) before running the program from the source code, as long as an interpreter can be installed on a machine, the program can then be run. This also drastically slow distributed package for programs as only source code must be distributed instead of compiled binary. However, the source code is leaked with all the security implication this can have. Finally, interpreted language usually have better memory management (built-in garbage collectors), are easier to debug, have very rich support of dynamic typing, dynamic scoping, reflections facilities, on-the-fly evaluation from the source code or even more like modifying the Abstract Syntax Tree (AST) resulting from the first compilation pass on source code by the interpreter. This last one is implemented in Common Lisp in the name of macros. There are more to say about interpreted languages, especially about those that are compiled into bytecode and tend to get the best of both worlds without the drawbacks, but this thesis will not discuss this matter any further.

The main point to understand here is that our main interest is set on C++, a compiled language with slow compilation time and very fast execution time. In C++ there are template metaprogramming to achieve genericity, but *templates do not generate any binary code*. Why? Because when the compiler meet a templated type or a templated routine, it does not know which type it will be instantiated with when it is used. Therefore, it can not compute information like type size, alignment, can not choose the right assembly instruction, for instance to compute an addition or a division (fixed vs. floating point arithmetic). This is why the compiler does not generate any binary code when it first meets templated code. The code is generated only when it is used with a concrete and known type. This is a huge problem. Now, if a library implementer wants to distribute his generic library, he must distribute source code and have the user to compile it. For a language like C++, with no standard dependency management, it can be a massive showstopper. Furthermore, it may not be reasonable for the user to have a C++ compiling infrastructure when the target machines are embedded devices with limited storage space. Indeed, C++ intermediary compilation artifacts tend to use a lot of disk space before they are linked into a smaller final binary. What solution do we have then?

SWILENA [21, 191] It is a Python bindings wrapper using Swig for the Olena C++ generic library. This wrapper enumerates all the common use cases and implement a binding for each of them. The compiler then generates binary code from the templated generic code for each use case enumerated in the wrapper. This way, we give to the dynamic world (Python) access to generic code (C++ template). However, it remains limited to the supported types. Each time a new combination of type needs to be supported from Python, it needs to be explicitly declared and compiled in the wrapper. Other image processing libraries, such as VIGRA [37], chose this solution.

VCSN [108] It is a novel solution that essentially takes the same base as SWILENA but goes beyond the boundary to implement a handmade facility that does system compiler calls to compile and link needed code on-the-fly when the binding does not exist. It then leverages the code hot loading feature of dynamic libraries to plug new dynamic libraries (.dll on Windows and .so on Linux) into the wrapper to provide the user with the requested bindings.

Cython [88] It attempts to solve the issue of the Python inherent language slowness due to its interpreted nature by providing a facility able to transpile a Python program into a C program so that a genuine C compiler (with extensions) is able to compile it and to link it against

the Python/C API in order to achieve an important performance gain at the cost of near zero knowledge of the complex Python/C API for the user. This novel solution essentially bypasses the work of a JIT compiler (that would be used by a programming language using bytecode such as Java or C#) and just offloads it onto a well known, proven and robust software: the machine's C compiler.

AutoWIG [141], Cppyy [130] and Xeus-cling [173] are all solutions aiming to automatically generate Python bindings on-the-fly using different solutions. AutoWIG has in-house code based on LLVM/Clang to parse C++ code in order to generate and compile a Swig Python binding using the Mako templating engine. Cppyy will generate Python bindings but can also directly interpret C++ code from Python code thanks to being base on LLVM/Cling, a Clang-base C++ interpreter. Finally, Xeus-cling is a Jupyter [129] kernel allowing to directly interpret C++ into a Jupyter notebook. Like Cppyy, it is based on LLVM/Cling. Those three projects are very promising and improve greatly the scope of possibilities for the future.

2.4 Summary

In this chapter we present the origin of generic programming, which goes as far as 1978 [1] and how it has evolved to be integrated in the Ada programming language and then the C++ programming language. Afterwards, it has evolved even further with the notion of *concept* which completes the toolbox required to be able to fully make use of generic programming without resorting to obscure tweaks and tools in C++.

This chapter explores the possibilities of achieving the notion of genericity from within a library. Indeed, there are three techniques enabling the user to write a high level algorithm once that can run on every type. They are the *code duplication* approach, the *generalization* approach and the *inclusion and parametric polymorphism* approach. We present in table 2.1 the result of the comparison of these approaches with regard to the features that we are interested in. We also discuss the limitations linked to the usage of those approaches by comparing OpenCV, Scikit-Image and Pylene, which make use of the four techniques at different level to achieve different goals. Furthermore, we have identified limitations related to the underlying data type, the structure of the domain, the optimizations and discuss the performances through a concrete benchmark presented in fig. 2.6.

This chapter also explores how the notion of genericity is achieved within programming languages. We retrace how CLU and then Ada implemented it and then how C++ permitted the expression of require-clauses (concept) as soon as C++98, even though it was limited at that time. We explore how template metaprogramming techniques have been developed and have evolved, alongside the C++ programming language itself, to finally reach a point in 2020 (C++20) where it is possible to write concepts in C++.

Finally, this chapter presents the inherent limitation of C++ templates: they must remain in the static world (compile-time). Genericity (in the sense C++ template) does not exist in the final shipped binary to the user. The final user, in its dynamic world (runtime), cannot use a generic (C++ code) tool. We discuss the different approaches possible to bridge this gap between the static (compile-time) and dynamic (runtime) world.

The next chapter will make extensive use of genericity to present the first contribution of this thesis: a taxonomy of concepts related to Image processing.

Part II

Applying Generic programming for Image processing in the static world

Chapter 3

Taxonomy for Image Processing: Image types and algorithms

In this thesis, we have researches how to apply all those new generic facilities from the C++ language into the Image processing area. This enables us to test them in a practical way on our predilection area while remembering our past work, both success and failures in this matter. However, as we saw in the previous Chapter (Generic programming (genericity)), birthing concepts from code is something that is done in an emerging way. Henceforth, the first work will be to do an inventory of all existing image algorithms as well as an inventory of all image processing algorithms (both basic and more complex) we can think of. This way, we will notice behavior patterns emerging from similar image types or similar algorithms. We will then be able to extract behavioral patterns from this inventory in order to produce a full taxonomy in the form of a framework of concepts related to image processing. This chapter is structured as followed. First we will study how to extract behavioral pattern from a simple algorithm in order to refine it into one or multiple concepts. Second we will study the theory set behind image types, their conjunctions, disjunctions. We will also produce an inventory of image processing algorithms limited to mathematical morphologies that we will leverage for the final step. Third, we will study the intrinsic genericity of algorithms to produce canvas taking advantage of properties (linked to the types). Finally, we will study behavioral patterns, related to the pre-established inventory of algorithms, in the form of a taxonomy engraved into a framework of concepts related to image processing.

3.1 Rewriting an algorithm to extract a concept

3.1.1 Gamma correction

Let us take the gamma correction algorithm as an example. The naive way to write this algorithm can be:

```
1  template <class Image>
2  void gamma_correction(Image& ima, double gamma)
3  {
4    const auto gamma_corr = 1.f / gamma;
5
6    for (int x = 0; x < ima.width(); ++x)
7      for (int y = 0; y < ima.height(); ++y)
8      {
9        ima(x, y).r = 256.f * std::pow(ima(x, y).r / 256.f, gamma_corr);
10       ima(x, y).g = 256.f * std::pow(ima(x, y).g / 256.f, gamma_corr);
11       ima(x, y).b = 256.f * std::pow(ima(x, y).b / 256.f, gamma_corr);
12     }
13 }
```

This algorithm here performs the transformation correctly, but also makes a lot of hypotheses. Firstly, we suppose that we can write in the image via the `=` operator (l.9–11): it may not be true if the image is sourced from a generator function. Secondly, we suppose that we have a 2D image via the double loop (l.6–7). Finally, we suppose we are operating on 8-bits range (0–255) RGB via `'.r'`, `'.g'`, `'.b'` (l.9–11). All those three hypotheses are unjustified. Intrinsically, all we want to say is "*For each value of `ima`, apply a gamma correction on it*". Let us proceed to make this algorithm the most generic possible by lifting those unjustified constraints one by one.

Lifting RGB constraint: First, we get rid of the 8-bits color range (0–255) RGB format requirement. The double loop become:

```
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);
```

By lifting this constraint, we now require the type Image to define a nested type `Image::value_type` (returned by `ima(x, y)`) on which `std::numeric_limits` and `std::pow` are defined. This way the compiler will is able to check the types at compile-time and emit warning and/or errors in case it detects incompatibilities. We are also able to detect it beforehand using a `static_assert` for instance.

Lifting bi-dimensional constraint: Here we need to introduce a new abstraction layer, the *pixel*. A *pixel* is simply a couple $(point, value)$. The double loop then becomes:

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

This led to us requiring that the type *Image* requires to provide a method `Image::pixels()` that returns *something* we can iterate on with a range-for loop: this *something* is a *Range* of *Pixel*s. This *Range* is required to behave like an *iterable*: it is an abstraction that provides a way to browse all the elements one by one. The *Pixel* is required to provide a method `Pixel::value()` that returns a *Value* which is *Regular* (see section 3.1.3). Here, we use `auto&&` instead of `auto&` to allow the existence of proxy iterator (think of `vector<bool>`). Indeed, we may be iterating over a lazy-computed view (cf. chapter 4).

Lifting writability constraint: Finally, the most subtle one is the requirement about the *writability* of the image. This requirement can be expressed directly via the new C++20 syntax for *concepts*. All we need to do is changing the template declaration by:

```
template <WritableImage Image>
```

In practice the C++ keyword `const` is not enough to express the *constness* or the *mutability* of an image. Indeed, we can have an image whose pixel values are returned by computing $cos(x+y)$ (for a 2D point). Such an image type can be instantiated as *non-const* in C++ but the values will not be *mutable*: this type will not model the *WritableImage* concept.

Final version

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

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

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

When re-writing a lot of algorithms this way: lifting constraints by requiring behavior instead, we are able to deduce what our *concepts* need to be. The real question for a *concept* is: "*what behavior should be required?*"

3.1.2 Dilation algorithm

To show the versatility of this approach, we will now attempt to deduce the requirements necessary to write a classical *dilate* algorithm. First let us start with a naive implementation:

```
1  template <class InputImage, class OutputImage>
2  void dilate(const InputImage& input_ima, OutputImage& output_ima)
3  {
4    assert(input_ima.height() == output_ima.height()
5      && input_ima.width() == output_ima.width());
6
7    for (int x = 2; x < input_ima.width() - 2; ++x)
8      for (int y = 2; y < input_ima.height() - 2; ++y)
9      {
10       output_ima(x, y) = input_ima(x, y)
11       for (int i = x - 2; i <= x + 2; ++i)
12         for (int j = y - 2; j <= y + 2; ++j)
13           output_ima(x, y) = std::max(output_ima(x, y), input_ima(i, j));
14     }
15 }
```

Here we are falling into the same pitfall as for the *gamma correction* example: there are a lot of unjustified hypotheses. We suppose that we have a 2D image (l.7–8), that we can write in the `output_image` (l.10, 13). We also require that the input image does not handle borders, (cf. loop index arithmetic l.7–8, 11–12). Additionally, the *structuring element* is restricted to a 5×5 window (l.11–12) whereas we may need to dilate via, for instance, a 11×15 window, or a disc. Finally, the algorithm does not leverage any potential properties such as the *decomposability* (l.11–12) to improve its efficiency. Those hypotheses are, once again, unjustified. Intrinsically, all we want to say is "For each value of `input_ima`, take the maximum of the $X \times X$ window around and then write it in `output_ima`".

To lift those constraints, we need a way to know which kind of *structuring element* matches a specific algorithm. Thus, we will pass it as a parameter. Additionally, we are going to lift the first two constraints the same way we did for *gamma correction*:

```
template <Image InputImage, WritableImage OutputImage, StructuringElement SE>
void dilate(const InputImage& input_ima, OutputImage& output_ima, const SE& se)
{
  assert(input_ima.size() == output_ima.size());

  for(auto&& [ipix, opix] : zip(input_ima.pixels(), output_ima.pixels())
  {
    opix.value() = ipix.value();
    for (const auto& nx : se(ipix))
      opix.value() = std::max(nx.value(), opix.value());
  }
}
```

We now do not require anything except that the *structuring element* returns the neighbors of a pixel. The returned value must be an *iterable*. In addition, this code uses the `zip` utility which allows us to iterate over two ranges at the same time. Finally, this way of writing the algorithm allows us to delegate the issue about the border handling to the neighborhood machinery. Henceforth, we will not address this specific point right now. This is seen in-depth later in section 4.4.

3.1.3 Concept definition

The more algorithms we analyze to extract their requirements, the clearer the *concepts* become. They are slowly appearing. Let us now attempt to formalize them. The formalization of the *concept Image* from the information and requirements we have now is shown in table 3.1 for the required type definitions and in table 3.2 for the required valid expressions.

Let *Ima* be a type that models the concept *Image*. Let *WIma* be a type that models the concept *WritableImage*. Then *WIma* inherits all types defined for *Image*. Let *SE* be a type that models the concept *StructuringElement*. Let *DSE* be a type that models the concept *Decomposable*. Then *DSE* inherits all types defined for *StructuringElement*. Let *Pix* be a type that models the concept *Pixel*. Then we can define:

	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 3.1: Concepts formalization: definitions

Let *cima* be an instance of *const Ima*. Let *wima* be an instance of *WIma*. Then all the valid expressions defined for *Image* are valid for *WIma*. Let *cse* be an instance of *const SE*. Let *cdse* be an instance of *const DSE*. Then all the valid expressions defined for *StructuringElement* are valid for *const DSE* Let *cpix* be an instance of *const Pix*. Then we have the following valid expressions:

	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
Structuring Element	`cse(cpix)`	`WIma::pixel_range`	returns a range of the neighboring pixels to iterate over it
Decomposable	`cdse.decompose()`	`implementation defined`	returns a range of structuring elements to iterate over it

Table 3.2: Concepts formalization: expressions

The *concept Image* does not provide a facility to write inside it. To do so, we have refined a second *concept* named *WritableImage* that provides the necessary facilities to write inside it. We say "*WritableImage* refines *Image*".

The *sub-concept ForwardRange* can be seen as a requirement on the underlying type. We need to be able to browse all the pixels in a forward way. This *concept* will not be detailed here as it is very similar to the standard library *concept* of the same name [193, 195]. Also, in practice, the *concepts* described here are incomplete. We would need to analyze several other algorithms to deduce all the requirements so that our *concepts* are the most complete possible. One thing important to note here is that to define a simple *Image concept*, there are already a large amount of prerequisites: *Regular*, *Pixel* and *ForwardRange*. Those *concepts* are basic but are also tightly linked to the *concepts* in the STL [194]. We refer to the STL *concepts* as *fundamental concepts*. *Fundamentals concepts* are the basic building blocks on which we work to build our own *concepts*. We show the C++20 code implementing those *concepts* in fig. 3.1.

```
template <class Ima>
concept Image = requires {
    typename Ima::value_type;
    typename Ima::pixel_type;
    typename Ima::const_pixel_range;
  } && Regular<Ima::value_type>
  && ForwardRange<Ima::const_pixel_range>
  && requires(const Ima& cima) {
    { cima.pixels() }
      -> Ima::const_pixel_range;
  };

template <class I>
using pixel_t = typename I::pixel_type;
template <class SE, class Pix>
concept StructuringElement = Pixel<Pix>
  && requires(const SE& cse,
       const pixel_t<Ima> cpix){
    { se(cpix) } -> Ima::const_pixel_range;
  };
```

```
template <class WIma>
concept WritableImage = requires Image<WIma>
  && requires {
    typename WIma::pixel_range;
  } && ForwardRange<WIma::pixel_range>
  && ForwardRange<WIma::pixel_range,
       WIma::pixel_type>
  && requires(WIma& wima) {
    { wima.pixels() } -> WIma::pixel_range;
  };

template <class DSE, class Pix>
concept Decomposable =
  StructuringElement<DSE, Pix>
  && requires(const DSE& cdse) {
    { cdse.decompose() }
      -> /*impl. defined*/;
  };
```

Figure 3.1: Concepts in C++20 codes

3.2 Image types viewed as Sets: version, specialization & inventory

Achieving true genericity in a satisfactory way is a complex problem that has components at different levels. The first goal is to natively support as many sets of image type as possible. Natively means that there is no need for a conversion from one type to a super-type under the hood. The second step is to support an abstraction layer above the underlying data type for each pixel. Indeed, the structure of an image is decorrelated from the underlying data type. The third step is to write image processing algorithms for each set of image type. The fourth step is that the performance trade-off shall be negligible if not null. The final step is to provide a high degree of friendliness to the end user. Ease of use must always be considered, as notation (as in writing code) is a tool of thought [74]. Indeed, "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race." [13].

After considering the available options to achieve our goal, the parametric polymorphism approach is the way to go. This enables the implementer to design image types and algorithms with behavior in mind. To illustrate this statement, let us consider the set of supported set of image types shown in fig. 3.5. We usually refer to different image families as subtype. Indeed, an image with a LUT is subtype of the (global) Image type. This notion of sub-typing is important because it may be abstracted behind an interface. A user can manipulate an image type without knowing its specific subtype. This induces that the sub-typing facility may be handled internally in the library with dynamic dispatching code (with runtime overhead). Each image processing library has its own way of handling sub-typing. More generally, the model used to handle it is previously described in section 2.1.1. Indeed, subtypes are handled the same way as their super-type with the fact that they have additional properties the algorithms can leverage.

There really are two distinct ways of implementing a basic image algorithm such as `fill`. For the set of images type whose values are encoded into each pixel, one must traverse the image and set each pixel's color to the new one. However, for the set of images type whose data type is encoded in a look-up table, one only has to traverse the look-up table to set each color to the new one. This translates into two distinct algorithms shown in fig. 3.2. We can represent the diagram outlining that those two algorithms are two distinct versions in fig. 3.5.

More generally, we consider that the set of image type is formed of several subsets of image types. In the example there are two subsets: images whose pixels are writable and images whose

$fill(I, v) : \forall p \in \mathcal{D}, I(p) = v$

(a) Writable image fill algorithm

$fill(I, v) : \forall i \in I.LUT, i = v$

(b) Image LUT fill algorithm

Figure 3.2: Comparison of implementation of the `fill` algorithm for two families of image type.

values are ordered in a look-up table. *For each one of these subsets, if there is a way to implement an algorithm then we have a* version *of this algorithm.*

Sometimes, it is possible to take advantage of a property on a particular image set that may be correlated to an external data to write the algorithm in a more efficient way. When those properties are linked to the types, it is called an algorithm *specialization* [68]. For instance, when considering a dilation algorithm, if the structuring element (typically the disc) is decomposable then we can branch on an algorithm taking advantage of this opportunity: decompose the disc into small vectors (radial decomposition [12] or periodic line decomposition [23]) and apply each one of them on the image through multiple passes. The speed-up comparing to a single pass with a large disc is really significant (illustrated in fig. 3.3). The code in fig. 3.4 illustrate how an algorithm can be written to take advantage of the structuring element's decomposability property. The algorithm will first decompose the structuring element into smaller 1D periodic lines. It will then recursively call itself with those lines to do the multi-pass and thanks to known optimizations on periodic lines [11], it will be much faster. The dispatch diagram outlining the different specialization of the dilation algorithm used is show alongside in fig. 3.4.

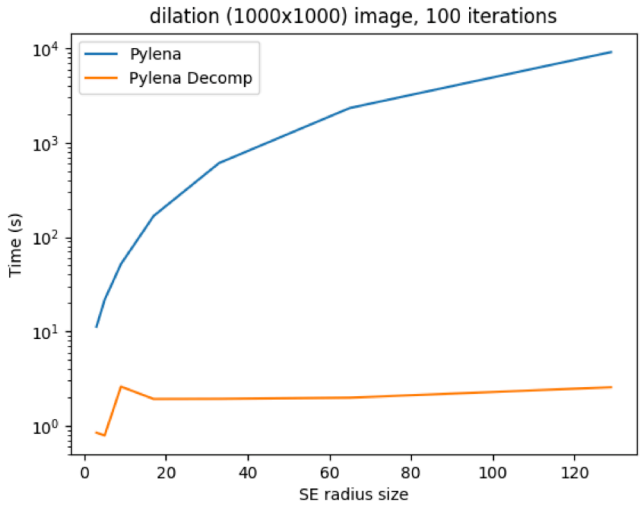


Figure 3.3: Benchmark: dilation of a 2D image (1000×1000) with a 2D disc (decomposable vs. non decomposable).

The fig. 3.5 shows how an algorithm specialization may exist in a set of algorithms version. There exists a specialization of algorithms when the data buffer is known to have the following property: its memory is contiguous. This implies that, for example, an algorithm like `fill` can be implemented using low level and fast primitives such as `memset` to increase its efficiency.

Making a full inventory of image types is not possible as there are many families of image types, each family may intersect with each other, images may have some particular properties at some points, those properties may appear across several families of image types. We can nonetheless cite a few to illustrate our points. There exists image types whose values are cubical complexes [52] or layered as hexagonal or triangular grid instead of square grids (such as meshes), that are represented as graphs [81], etc. All of those interleaves with different research areas (computer vision for meshes, topological classification for complex cells [164, 41], graph theory

```
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)
      img = dilate(img, small_se) // Recursive call
    return img;
  } else if (is_pediodic_line(se))
    return fast_dilate1d(img, se) // Van Herk's algorithm;
  else
    return dilate_normal(img, se) // Classic algorithm;
}
```

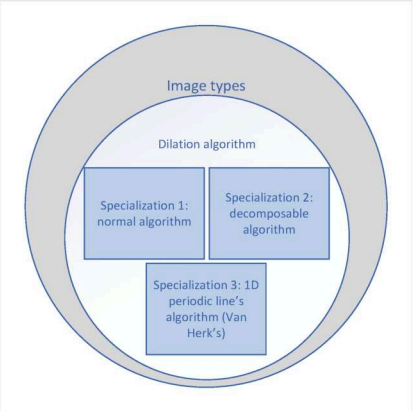
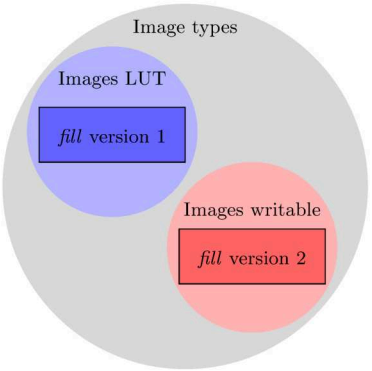
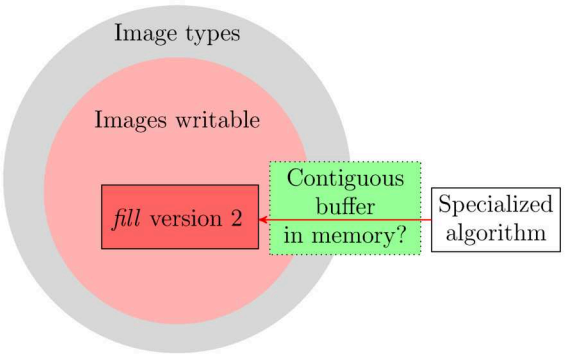


Figure 3.4: Dilate algorithm (left) with decomposable structuring element and its specialization diagram (right).



(a) Different versions of *fill* algorithm



(b) Specialization existing within a version

Figure 3.5: Set of algorithm version (a) and its specialization existing within a version (b).

and hierarchy for image graphs [113, 121, 153]). Finally, this image type inventory does not help when it comes to designing an image processing library. Instead, what helps is making an inventory of image processing algorithms as well as what properties those algorithms can leverage to speed up their execution.

Therefore, making the inventory of image processing algorithms (limited to mathematical morphology) outlines three main families of image processing algorithms. The first one regroups the pixel-wise algorithms. Those algorithms are the basic pixel-wise transformation. They are the base of any image processing toolkit. Indeed, being able to extract the green or red color channel of an image is mandatory. Thresholding as well as the previously seen gamma correction belongs to this family. The second one regroups the local algorithms. Those algorithms perform transformations by considering all the neighboring pixels around a given pixel. Those algorithms need additional data in the form of an image's extension (to define border's behavior) and a structuring element (disc, rectangle around the considered pixel). Such algorithms are widely used in mathematical morphology. Dilation, erosion, closing, opening, gradient, rank filter are all local algorithms, as well as stencil-type algorithms such as union-find, max-tree or skeletonize. Finally, the third family regroups the global image processing algorithms. This set contains algorithms where computing the current pixel requires to knowledge about what was previously computed for the previous pixels. Chamfer distance transform, labeling, watershed, hierarchy structures related algorithms belongs to this category.

3.3 Generic aspect of algorithm: canvas

Genericity is always referred to with this sentence "write once, work for every type, run everywhere". However, very quickly we learned that the *run* aspect can be a combination of:

- as fast as possible on a single CPU unit;
- as fast as possible by using many CPU units;
- as fast as possible by using many GPU units;
- as fast as possible by using many computers (cloud) and their CPU/GPU units.

How do we decide what is the most efficient way to do? There is no simple answer to this question, but we can start by studying the computational shape of algorithms with two goals in mind: (i) find what can be parallelized/distributed and (ii) find one or several algorithms' abstraction. Indeed, in image processing there are a lot of common patterns when one looks at algorithms implementations, the most famous being *for all pixels of an image, do something to each pixel*. But there are other more high-level similarities that we can leverage to have more generic algorithm. Let us first study the different programmatic models there are commonly used to process images.

The pipeline This old and proven model is especially useful to do pre-processing and post-processing routines. It is a very robust way of structuring a workflow. It consists in an imbrication of different operators (algorithms) taking as input one or several images, maybe additional data (such as labels, adjacency map, etc.) in order to process the data int a "left-to-right" order. The result will show at the end of the pipeline and the optimization opportunities are located inside the smaller operators and in the form of correctly managing data (no unnecessary data copy, locality, etc.)

Kernels and tiling This programmatic model is engineered to leverage the massive resources available on GPU. This is the trendy way of the last decade. It consists in breaking the original image into small tiles, and then to feed those tiles into a massively multicore GPU (via CUDA,

Halide [112]). Processing will happen concurrently on those core, but it is costly to swap contents inside those GPU from the RAM memory. It's then preferred to design pipeline for those tiles so that the work directly happens on each core to minimize the number of back and forth copies from the RAM.

Cloud computing It is a huge deal in image processing these last years as well, especially used in Deep learning applications. Deep neural networks are usually executed on cluster of computer through network (cloud) and thus are leveraging combinations of successive *MapReduce* [77] under the hood. Deep learning frameworks (such as Tensorflow/Keras [124, 122, 134], PyTorch [151], etc.) take care of offloading work locally (CPU multithreading, GPU kernels and tiling) while cloud provider infrastructures (such as Azure, AWS) dispatch the top level tasks on their clusters.

In every case there is a notion of pipeline where the user pipe algorithms into each others in order to achieve a result. Those algorithms can leverage all the heterogeneous resources (cloud, GPU, CPU) available to *map* the input data. Algorithms will then finally aggregate the results (*reduce*) to output them into another algorithm, or save them, or display them. It is important to dissociate the route the data will go through and the processing pipeline logic. Both have their own specificities. In this thesis, we make a parallel, at small scale, between processing pipeline logic and image processing algorithms. First let us study two basic algorithms: dilation and erosion. The Python code for those algorithms is naively given in fig. 3.6.

```
def dilate(img, se, out):
  for pnt in img.points():
    out(pnt) = img(pnt)
    for nx in se(pnt):
      out(pnt) = max(out(pnt), img(nx))
```

(a) Dilation

```
def erode(img, se, out):
  for pnt in img.points():
    out(pnt) = img(pnt)
    for nx in se(pnt):
      out(pnt) = min(out(pnt), img(nx))
```

(b) Erosion

Figure 3.6: Dilate and Erode algorithms.

The algorithms are almost written the same way. The only change is the operation *min* and *max* when selecting the value to keep. As such, we can easily see a way to factorize code by passing the supremum operator as a function parameter. The algorithms can then be rewritten as shown in fig. 3.7. In this last example, we have one piece of code which is in charge of abstracting the way an image is traversed: we name it the *canvas*. We also have other pieces of code which carry the logic of the operations, calling the canvas and providing the logic to apply from within the canvas. This way of decomposing the code offers the opportunity of writing more specific heterogeneous logic for the traversing code so that the other parts of the code that carry the high-level logic remains unaware and unburdened of possible implementation details.

```
def local_op(img, se, op, out):
  for pnt in img.points():
    out(pnt) = img(pnt)
    for nx in se(pnt):
      out(pnt) = op(out(pnt), img(nx))
```

(a) Local algorithm with custom operator

```
def dilate(img, se, out):
  local_op(img, se, max, out)
```

(b) Dilation (delegated)

```
def erode(img, se, out):
  local_op(img, se, min, out)
```

(c) Erosion (delegated)

Figure 3.7: New Dilate and Erode algorithms.

3.3.1 Taxonomy and canvas

This approach is very much compatible with the inventory of the algorithms we did in the previous section 3.2. Indeed, for the point-wise family of algorithms, it is hardly an issue as they

can all be written in terms of views (cf. chapter 4). For the second family that consists in all the local algorithm, they can be abstracted behind an algorithm canvas where the user provides the task to perform for each point of the algorithm. For instance, a single pass local algorithm will always have shape given in fig. 3.8. Finally, for the third family which consists in all the algorithm that propagate their computation while traversing the image. There is no trivial way to provide an algorithmic canvas as each algorithm has its own way to propagate changes curing its computation.

```
1  def local_canvas(img, out, se):
2    # do something before outer loop
3    for pnt in img.points():
4      # do something before inner loop
5      for nx in se(pnt):
6        # do something inner loop
7      # do something after inner loop
8    # do something after outer loop
```

Figure 3.8: Local algorithm canvas.

The canvas at fig. 3.8 can be customized to do a specific job, especially at the lines 2, 4, 6, 7 and 8. The user would then provide callbacks and the canvas would perform the tasks. This is especially useful when knowing that the canvas can handle the border management (the user would only need to provide a handling strategy like mirroring the image or filling the border with a value). The canvas can also take advantage of optimization opportunities (such as decomposing a structuring element) that the user would probably forget, or not know, when first writing his local algorithm. Another advantage is the opportunity to do more complex optimization such as parallelizing the execution or offloading part of the calculation on a GPU. More generally, all optimization done through heterogeneous computing would be available by default even if the user is not a field expert.

Despite all these advantages, one big disadvantage is the readability of the algorithm user-side. For instance, the dilation algorithm leveraging the local canvas is rewritten in fig. 3.9.

```
def dilate(img, out, se):
  do_nothing = lambda *args, **kwargs: None

  def before_inner_loop(img, out, pnt):
    out(pnt) = img(pnt)

  def inner_loop(ipix, opix, nx):
    out(pnt) = max(out(pnt), img(nx))

  local_canvas(img, out, se,
    before_outer_loop = do_nothing,
    before_inner_loop = before_inner_loop,
    inner_loop        = inner_loop,
    after_inner_loop  = do_nothing,
    after_outer_loop  = do_nothing
  )
```

Figure 3.9: Dilation using the local algorithm canvas.

This way of thinking algorithms is far less readable than the classic way. The user does not see the loops happening, and it can become very messy when several passes are happening (closing, opening, hit or miss, etc.)

3.3.2 Heterogeneous computing: a partial solution, canvas

One of the key aspect driving genericity is performance. We still have the following mantra: "write once, work for every type, run everywhere". However, when considering the *run* aspect,

there is a lot to do. Indeed, nowadays, leveraging the available resources to their maximum is long-standing issue. There are many ongoing works on the subject, such as SyCL [147, 146] which is a standard for heterogeneous computing model edited by Kronos. This standard currently has four implementations: Codeplay's ComputeCpp [180], Intel's LLVM/clang implementation [181], triSYCL [183] led by Xilinx and hipSYCL [157] led by Heidelberg University. There also exists smaller libraries such as Boost.SIMD [114] or even VCL [109] to ease writing SIMD code. After taking some distance to study the subject, we can infer that there are three main aspects to consider when optimizing performance.

The first one, the most important one is the algorithm to use in accordance with the set of data. This aspect is covered by the C++ language and its built-in genericity tool: template metaprogramming. Indeed, we can select the most suitable algorithm for a particular set of data.

The second one is the ability for the code to be understood by the compiler so that it can be further optimized during the generation of the binary. Indeed, when compiling for the native architecture of a recent processor, the compiler can use the most recent assembly instructions to use wide vectorized registries (AVX512). The use of an up-to-date compiler infrastructure also brings the help much needed.

Finally, the third aspect is not as trivial as the first two ones. It consists in studying the structure of an algorithm to allow distributed computation. It also consists in studying the different architectures to select the most efficient algorithm for each particular architecture. Sometimes algorithm are friendly to their distribution on several processing units that compute a part of the result concurrently. This is what we call parallelism. There exists several ways to take advantage of parallelism. First there is the use of several CPU units on the host computer. Then there is the use of GPU units working in combination with the CPU units to take advantage of the massive amount of core a graphic card can provide. Finally, there is the use of cloud computing which consists in using several "virtual" computers, each of them offering CPU and GPU units available to compute a result. However, each time we introduce a new layer of abstraction, there is a cost to orchestrate the computation, send the input data and retrieve the results. Therefore, it is very important to study case by case what is needed. Some solutions exist to abstract away completely the hardware through a DSL [1] such as Halide [112]: the DSL compiler's job will be to try very hard to make the most out of both the available (or targeted) code and hardware. Those solutions are not satisfactory for us as we want to avoid using a DSL to remain at code level (core language only). We are not developing a compiler: we are working with it.

There is one true issue when studying parallel algorithms: it is whether they can be parallelized or not. Not all algorithms can be parallelized. Some just intrinsically cannot, typically, algorithms that immediately need the result of the previous iteration to compute the next iteration. There may exist specific ways to re-arrange a specific algorithm, for instance, taking advantage of some algebraic property, in order to rewrite it in a parallelizable way, but it is not trivial (it is done on a per-algorithm basis), and not generic. As such, it is out of the scope of this thesis. What interests us are the algorithms whose structure is an accumulation over a data type that can be defined as a monoid. We assert that every algorithm that can be rewritten as an accumulation over a monoid can be parallelized and/or distributed. This model, that consists in distributing computation like an accumulation over a monoid data structure, is also call the "*map-reduce*". This model has two steps: the distribution (map) and then the accumulation (reduce).

The *map* step dispatches computation on subunits with small set of data. The *reduce* step retrieves and accumulate all the results' data as soon as they are ready.

The accumulation algorithm has this shape:

```
1  template <class In, class T, class Op>
2  auto accumulate(In input, T init, Op op)
3  {
4    for(auto e : input)
5    {
```

[1] Domain Specific Language

```
6         init = op(init, e);
7       }
8       return init;
9     }
```

The loop line 4 can be split into several calculation units which are going to be distributed, and then accumulated later once the units have finished their computation.

The issue left here is the monoid. What exactly is a monoid here? A monoid is a data structure which operates over a set of values, finite or infinite. This data structure must provide a binary operation which is closed and associative. Finally, this data structure must also provide a neutral element (aka the identity). Some trivial monoids comes to mind:

- *boolean*. For binary operation "and", identity is "true" whereas for binary operation "or", identity is "false".
- *integer*. For binary operation $-$ and $+$, identity is 0 whereas for binary operation $*$ and $/$, identity is 1.
- *string*. For concatenation, identity is empty string.
- *optional* value (also known as monadic structure in Haskell programming language).

There are many more monoids, less trivial but very handy, such as the unsigned integer/max/0 set and the signed integer/min/global max set.

This theory is extremely beneficial to image processing as the most commonly used algorithms, the local algorithms, can all be written in the form of an accumulation over the pixels of an image. The fact that finding an identity for the operation processed by the algorithm is often quite trivial led us to the idea of canvas. A canvas is a standard way to write an iteration over an image which abstracts the underlying data structure. A canvas is a tool for the user to provide its computation model based on events such as: "entering inner loop" or "exiting inner loop". The user can then provide its operations as if he was writing his algorithm himself (restricted to the accumulation model). As the maintainer of the library provides the canvas of execution, he can know also make change to take advantage of it. For instance, computing a CUDA kernel at one point and dispatching it on GPU units is totally within scope and remains transparent for the user of the library. Although there is a caveat: rewriting our algorithm in an accumulation shape and chunking it in fragments to feed to the canvas is definitely not intuitive. Indeed, we require our user to change his way of thinking from the procedural paradigm to the event-driven paradigm. This approach is not new and is used in other libraries such as Boost.Graph [44] for similar purposes. Dean talks about this recurring monoid pattern more in-depth in his talk [148].

In image processing, we quickly identify *local* algorithms. They reason about a group of pixel around a given of coordinate. All those algorithms can be abstracted behind an accumulation of some sort, and they all have the same morphology. Thus leading to the following abstraction:

```
template <class In, class Out, class SE, class T, class Op>
auto local_accumulate(In input, Out output, SE se, T init, Op op)
{
    auto zipped_imgs = ranges::view::zip(input.pixels(), output.pixels())
                                                    // (1)
    for(auto&& rows : ranges::rows(zipped_imgs))
    for(auto [px_in, px_out] : rows)
    {
        auto v = op(init, px_in.val());             // (2)
        for(auto nb : se(px_in))
        v = op(v, nb.val());                        // (3)
        px_out.val() = v;                           // (4)
      }
                                                    // (5)
  }
```

From this code we can deduce some very useful and easy monoids by the following triplets:

- $(type = boolean, operator = and, neutral = true)$ is a binary erosion
- $(type = boolean, operator = or, neutral = false)$ is a binary dilation
- $(type = unsigned\ integer, operator = max, neutral = 0)$ is a dilation
- $(type = signed\ integer, operator = min, neutral = globalmax)$ is an erosion

Now if we want to rewrite the `local_accumulate` in an event driven paradigm, we need to identify the different callbacks to expose to our user on the call site. Especially, what will be the callback parameters. There are five callback events that we have identified:

1. before entering outer loop (no work is done)
2. before entering inner loop (iteration over the pixel's neighbor)
3. inner loop (actual operation to perform, result is accumulated)
4. after exiting inner loop (iteration over the neighbor is over, what to do with the accumulated result?)
5. after exiting outer loop (iteration over the image is over)

```
template <class In, class Out, class SE, class T, class BeforeOuterLoopCB,
          class BeforeInnerLoopCB, class InnerLoopCB class AfterInnerLoopCB,
          class AfterOuterLoopCB>
auto local_accumulate(In input, Out output, SE se, T init,
                      BeforeOuterLoopCB bolCB, BeforeInnerLoopCB bilCB,
                      InnerLoopCB ilCB, AfterInnerLoopCB ailCB,
                      AfterOuterLoopCB aolCB)
{
  auto zipped_imgs = ranges::view::zip(input.pixels(), output.pixels()
  bolCB(input, output);                              // (1)
  for(auto&& row : ranges::rows(zipped_imgs))
    for(auto [px_in, px_out] : rows)
    {
      bilCB(px_out.val(), init, px_in.val())         // (2)
      for(auto nb : se(px_in))
        ilCB(px_out.val(), px_out.val(), nb.val())   // (3)
      ailCB(px_out.val(), init, px_in.val())         // (4)
    }
  aolCB(input, output)                               // (5)
}
```

In this code, we can see that all the callbacks do not take the same type and/or number of parameters. Here is what the call site could look like if the user wants to perform a dilation:

```
local_accumulation(
  input,                                    // input image
  output,                                   // output image
  se,                                       // structuring element
  0,                                        // monoid's neutral element
  [](auto I, auto& O) { /* do nothing */ }, // (1) entering outer loop callback
  [](auto& o, auto init, auto in){          // (2) entering inner loop callback:
    o = std::max(init, in);                 // initialize with neutral element
  },
  [](auto& o, auto cur, auto nbh) {         // (3) inner loop callback:
    o = std::max(cur, nbh);                 // keep the local maximum
  },
  [](auto& o, auto init, auto in) {         // (4) exiting inner loop callback
    /* do nothing */
  },
  [](auto I, auto& O) { /* do nothing */ }  // (5) exiting outer loop callback
);
```

It is very verbose and non-intuitive but hopefully, once the compiler optimize out the empty callbacks, the generated code is as fast as a non-generic handwritten dilation.

3.4 Library concepts: listing and explanation

Let us now delve into the concepts related to the Image Processing area. Indeed, this domain has his specificities, and we want to improve generic image processing library design by learning from our past experiments and by working with new techniques. The most basic usage of an image is the famous algebraic formula $y = f(x)$ where y is a *value* generated by the *image* f for the *input* x. Aside from generating a value, an image can also *store* a value, as in $f(x) = y$ where the value is *assigned* to the image for a given point. Those notions are the basis of our work and will drive the entire design.

3.4.1 The fundamentals

First, let us introduce the fundamental concepts deriving from the basis notion. The *Value* concept is refined into three distinct one (details in appendix C.1.1). There are the basic *Value* but also the *ComparableValue* and the *OrderedValue* which are useful when it comes to comparison or ordering algorithms. The need behind those three concepts derives from the algorithms which require an ordering relation in order to function properly. Most of mathematical morphology algorithms requires it. For instance, the ordering relation for a gray-scale image is trivial whereas it is a field of research for colored (rgb 8-bits) images.

The second fundamental brick is the concept of *Point*, detailed in appendix C.1.2 which is a bit less open than the concept *Value* as it must be totally ordered. Indeed, when it comes to accessing a value stored in an image, whether it is for reading or for mutating purpose, it is important that there is exactly one value accessed.

We Now introduce an abstract way to represent this relation $Value \times Point$: the *Pixel*. This is a well known notion in image processing, and it represents a couple $(point, value)$. This abstraction layer is easy to move around and contains facilities to read and mutate the pixel's value if possible. Indeed, not all pixels can mutate their value. If the pixel is yielded by an image that only generates values on the fly then it cannot mutate it. Henceforth, we introduce two new concepts: *Pixel* and *OutputPixel* (details in appendix C.1.3). Those two concepts have a very similar interface described in appendix C.1.3. They can both access the stored information: the point and the value. On top of that, the *OutputPixel* can mutate its value. When interacting with pixels, the user is able to write code as followed:

```
auto pix = Pixel();     // Get a pixel
auto val = pix.val();   // yield the pixel value
auto pnt = pix.point(); // yield the pixel point
pix.val() = 42;         // Assign a value
```

We show how those three fundamental concepts interact with each other in the diagram fig. 3.10.

Now we need a helper concept: the *ranges*. Ranges [184, 143, 195, 193] are a set of concepts defined in the C++ standard library shipped with the ISO C++20 norm in 2020 [162]. They allow the user to abstract away iterators so that the iteration occurs directly over the values. This offers the user the opportunity to migrate his source code from being:

```
template <class IteratorBegin, class IteratorEnd>
void my_algorithm(IteratorBegin beg, IteratorEnd end)
{
  for(; beg != end; ++beg)
    // use *beg to access the value
}
```

To being:

```
template <class Range>
void my_algorithm(Range rng)
{
  for(auto&& val : rng)
    // use val  to access the value
}
```

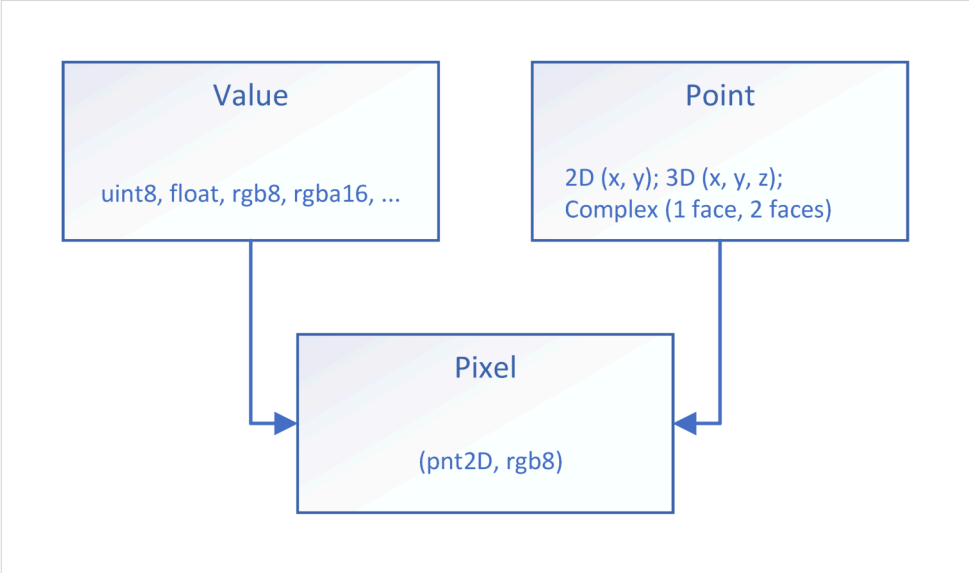


Figure 3.10: Pixel concept.

In image processing, we refine further this concept by introducing multidimensional ranges (*MDRange*). Indeed, in image processing the user is used to write double loop to iterate over a bi-dimensional image. Abstracting away this aspect under standard ranges induces performance loss. That is why we need to introduce this additional concept. A multidimensional range can be split with a library function, `mln::ranges::rows(mdrng)` to fit the double-loop pattern while retaining high performances. This topic is tackled in-depth later in section 4.5.1. For now, let us consider multidimensional ranges as an image processing extension for performance for the image traversing pattern. They are defined in appendix C.1.4 and their interface is the same as standard ranges, as seen in appendix C.1.4. They are designed so that the user code can look like this:

```
auto mdrng = MDRange(); // Get a multi-dimensional range of values
auto rows = mln::ranges::rows(mdrng);
for(auto row : rows) // double loop pattern
  for(auto val : row)
    // use(val)
```

From an algebraic point of view, the definition of an image is not complete without considering a definition domain on which it is defined. In image processing, the same rule applies. We cannot consider an image without considering the set of points that are valid for this image. Henceforth, we must define the concept of *Domain* (details in table C.7). The *Domain* concept is refined into two sub-concepts named *SizedDomain* and *ShapedDomain*. This makes the existence of infinite domain and domains that may be defined over non-continuous intervals in space possible. This enables algorithms to require the domain to have certain shape if needed. The domain behavior is described in appendix C.1.5.

In practice, a domain is used to get information about the points constituting the image. Indeed, we can write code like this:

```
auto dom = Domain(); // Get a domain
auto pnt = Point(..., ...); // Get a random point
bool ret = dom.has(pnt); // Check wether the domain contains the point
bool is_empty = dom.empty(); // Check wether the domain is empty
auto dim = dom.dim(); // Yield the domain's dimension information
for(auto pnt : dom) // browse the definition domain
  // use pnt as a point of the domain
```

We show how the concept *Domain* flows from the previous concepts in the diagram shown in fig. 3.11.

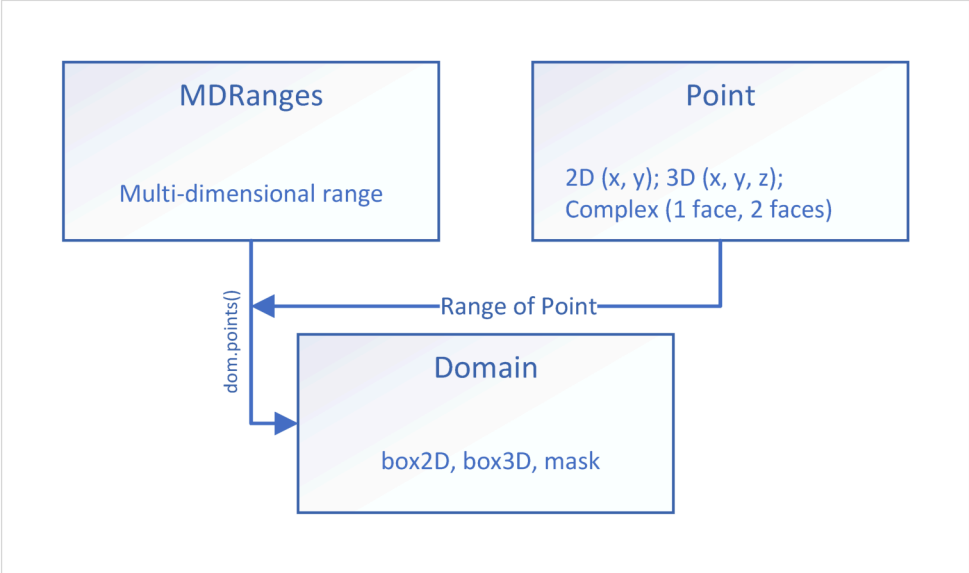


Figure 3.11: Domain concept.

All the prerequisites to introduce our main concept: *Image* are fulfilled. Similar to *Pixel*, we have the distinction over image whose value can be mutated in a sub-concept named *WritableImage*. These concepts are defined in table C.9. In addition to this definition, we can infer the behavior described in appendix C.1.6. There are complicated requirements written in template metaprogramming code and, in summary, they just require that the value types of the ranges returned by the member functions `pixels()` and `values()` are the same as the value types declared in the parent image type. In addition, we introduce two facilities which are the member function `concretize()` and `ch_value<V>()`. The first is a way to turn a view into a concrete type. This will be seen more in-depth in chapter 4. The last is a way to cast values from one type to another. It forms a new image type whose underlying values will be returned after being converted to a new value type. This last facility is extremely useful when we only want to cast the underlying type while keeping all the other details (such as the dimension) identical. For instance, when working with labeling algorithm, we know our algorithm will return an image similar to the input one except for the underlying type which will be the type of the label. The following code shows how it is used:

```
using label_t = int; // label type

template <class I> // Input image of type I
auto my_labeling_algorithm(I input_image)
  -> image_ch_value_t<I, label_t> // Output image is Input image (I)
                                  // whose underlying type is label_t
{
  // ...
}
```

We show the diagram building up the image concept from the previous concepts in fig. 3.12.

3.4.2 Advanced way to access image data

While being able to iterate over ranges of pixels or values is good, we are still lacking fundamental facilities to access an element directly from the image. In order to solve this predicament, first we need to define the concept of *Index* in appendix C.2.1 which we will use afterwards. This is a very simple concept that encapsulate an integral value. This value can be negative as we want to be able to do negative indexing in case our image has an extension, cf. section 3.4.2. The first advanced concept is represented as an *IndexableImage*. An element can be accessed simply

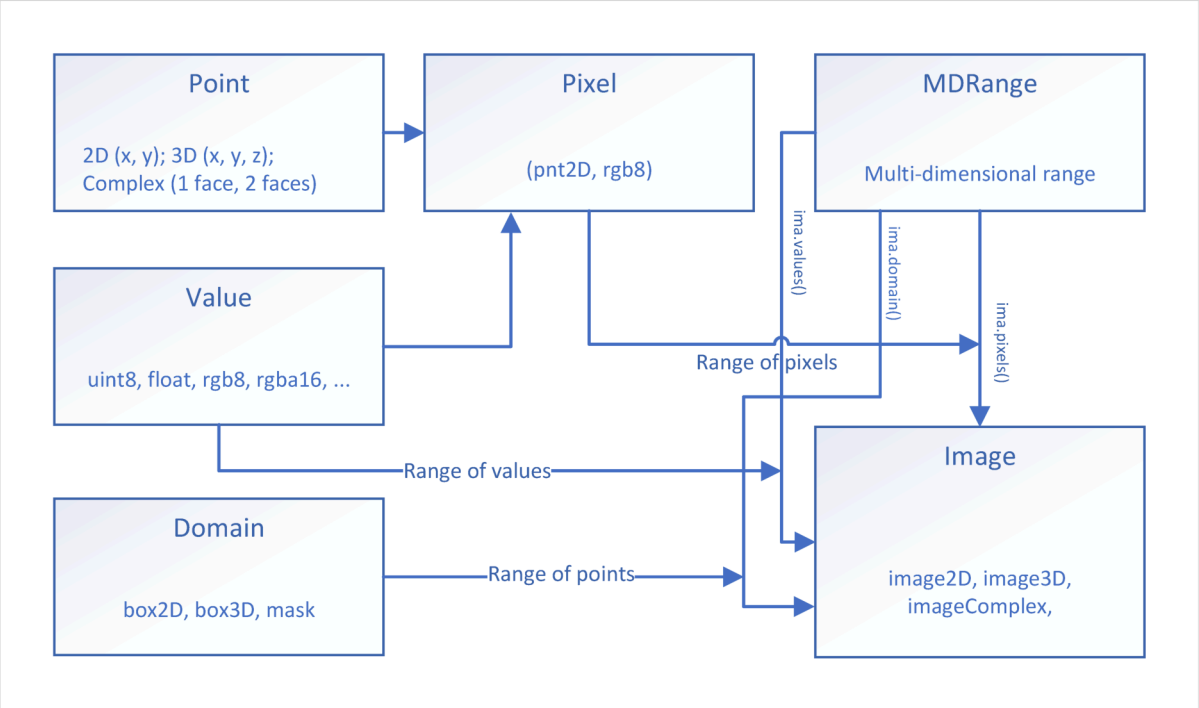


Figure 3.12: Image concept.

by providing its index number. This concept is defined in table C.12. This introduces a simple behavioral pattern described in appendix C.2.2. With this concept, we are able to write code as followed:

```
auto ima = IndexableImage(); // Get an indexable image
int k = 15; // Get an index
auto val = ima[k]; // Get the element at the index
ima[k] = 255; // Mutates the element at the index
```

Being able to traverse an image through indexes is especially useful for algorithms that are aware of the number of elements in the image. We chose to be flexible with our indexing method (i.e. allowing signed indexes) not to fall into the same pitfall as the C++ standard library [196]. Indeed, requiring that the standard type `std::size_t` is unsigned led to loads of issues, the first one being a conversion issue when writing a simple for-loop. Solving these issues led to the appearance of new member function `.ssize()` (for signed size) and the new standard type `std::ptrdiff_t` to store the result of a subtraction between two `std::size_t`. Furthermore, in our specific area (image processing), it may be well-defined to access a buffer with negative indexes when, for instance, we are accessing the value of the extension of an image. This is why our indexes are signed.

Additionally, we want to be able to access a value by providing a point, the same way as in the algebraic definition $val = image(point)$. To do so, we introduce the concept of accessibility through *AccessibleImage*. This concept is defined in table C.14. This introduces new behavior that is described in appendix C.2.3. We can notice some facilities specifically including bound checking. Indeed, we suppose, for fast access, that the user is always picking element from the image's domain, but it is possible to bound check elements if needed on access for specific usages. With this concept, we are now able to write code as followed:

```
auto ima = AccessibleImage(); // Get an accessible image
auto p = Point(); // Get a point
auto val = ima(p); // Get a value from a point
auto val = ima.at(p) // Same with no bound checking
auto pix = ima.pixel(p) // Get a pixel from a point
auto pix = ima.pixel_at(p) // Same with no bound checking
```

```
ima(p) = 42; // Assigns a value from a point
ima.at(p) = 42; // Same with no bound checking
ima.pixel(p).val() = 42; // Assigns a pixel value from a point
ima.pixel_at(p).val() = 42; // Same with no bound checking
```

Being able to traverse an image with points is especially useful for algorithms relying on restricting/expanding definition domain that are exclusively yielding points.

Once we know that an image is both *indexable* and *accessible* we can introduce new behaviors (described in appendix C.2.4) that we put behind the concept of *IndexableAndAccessibleImage* defined in table C.16. This behavior is related to retrieving an index from points and vise versa. Indeed, it is now possible to write such code:

```
auto pnt = ima.point_at_index(k); // Get the point from an index
auto k = ima.index_of_point(pnt); // Get the index from a point
// Get the index difference for a shift of delta_point
auto delta_idx = ima.delta_index(delta_pnt);
```

Additionally, it is useful, for propagating algorithms, to be able to traverse images in both a forward way and a backward way. As it may not be possible for all images, this notion needs to be refined into a new concept *BidirectionalImage*. This concept is defined in table C.18 and its behavior is described in appendix C.2.5. Thanks to this concept, we are able to write code as followed:

```
template <class I>
my_algorithm(I input)
{
  // forward pass
  for(auto pix : input.pixels())
    // use pix

  auto backward = mln::ranges::views::reverse(input.pixels())
  for(auto pix : backward)
    // use pix
}
```

Finally, we need a way, when possible, to iterate over a contiguous data buffer for very fast and optimized calculation. That is what the concept of *RawImage* is for: an image whose data buffer can be accessed, as well as its mutable counterpart, which are defined in table C.20. Having a raw image whose data buffer can be accessed enables us to expose two more member functions to access the data buffer and its strides to compute accurate pointer arithmetic. They behave as described in appendix C.2.6. This allows writing code as followed:

```
auto ima = Image(); // Image of int
const int* data = ima.data(); // Access the underlying buffer
auto dim = ima.domain().dim(); // Get the dimension of the image
// Retrieve information about strides
auto strides = std::vector<std::ptrdiff_t>(0, dim)
for (int i = 0; i < dim; ++i)
  strides[i] = ima.stride(i)

// Now use data and strides to traverse the raw buffer
// ...
```

We show how those concepts are defined and interact with each other in the diagram shown in fig. 3.13.

3.4.3 Local algorithm concepts: structuring elements and extensions

From the beginning concepts emerge from behavioral patterns extracted from algorithms. In image processing, there is a family of algorithms called the *local algorithms*. They work by considering a specific pixel as well as all the pixels surrounding within a window that has a specific shape centered in this first pixel. The window is called the *structuring element* and the pixels considered by this window are called the *neighborhood*. This leads us to introduce the concept of *StructuringElement* which is defined in table C.22.

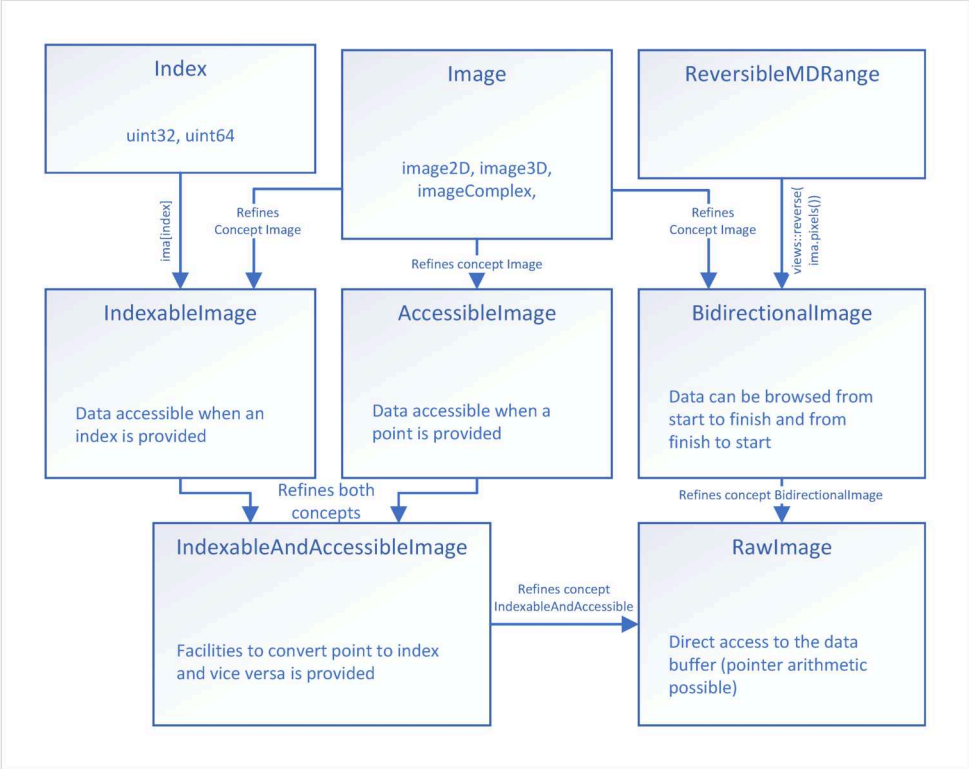


Figure 3.13: All images concepts.

This concept is refined into three sub-concepts that are related to properties a structuring element can provide. Those properties are:

- decomposability: ability to split a complex structuring element into several smaller and simpler structuring element. There is an equivalence in behavior when the algorithm is recursively run for each smaller structuring element one after another, in a multi-pass way.
- separability: ability to split a complex structuring element into several smaller and simpler structuring element. There is an equivalence in behavior when the convolution is recursively run for each smaller structuring element one after another, in a precise order, in a multi-pass way.
- incremental: ability to tell the points that are added to or removed from the range when the structuring element is shifted by a basic displacement (e.g. for a *2D point*, the basic displacement is $(0, 1)$). Usually used to compute attributes over a sliding structuring element in linear time.

The fig. 3.14 shows how a $5x5$ rectangle is decomposed into periodic lines. The fig. 3.15 shows how a disc of radius 3 is decomposed in periodic lines.

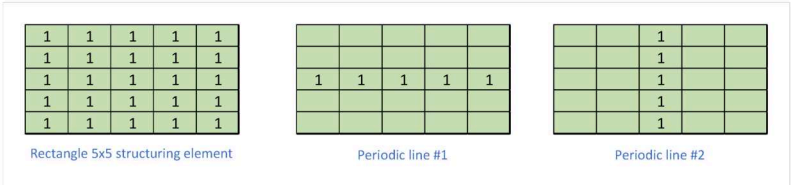


Figure 3.14: Decomposition in period lines of a rectangle structuring element.

The behavior requirements of those concepts is described in appendix C.3.1. Being able to manipulate structuring element allows us to write the following code:

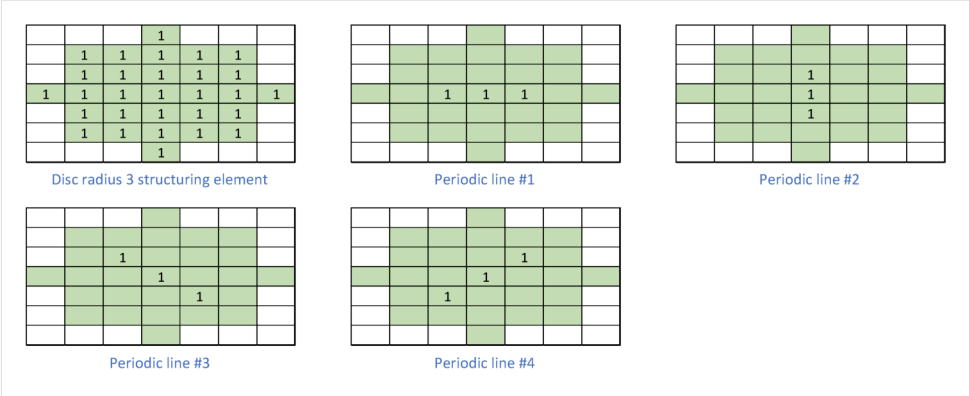


Figure 3.15: Decomposition in period lines of a disc structuring element.

```
auto se = se::disc(.radius=3); // get a structuring element
for(auto pix : ima.pixels())   // traverse image
  for(auto nb : se(pix))       // traverse neighboring pixels
    // ...
```

Additionally, we introduce the concept of *Neighborhood* in table C.24. This concept has facilities to know what points/pixels are placed before or after another point/pixel inside the window of a specific structuring element. It behaves as described in appendix C.3.2. This concept is useful when one wants to only consider a certain part of the neighboring pixels within a structuring element. This offers the opportunity to write the following code:

```
auto se = se::disc(.radius=3); // get a structuring element
for(auto pix : ima.pixels())   // traverse image
  for(auto nb : se.before(pix))// traverse neighboring pixels located before pix
    // ...
```

And the last concept we need to introduce is the *extension*. Indeed, extension management is very important when dealing with local algorithm as pixels on the border need to be processed too. Indeed, the behavior near the border of the image must be defined and well-specified. There are several strategies when it comes to borders and extension. We refine a concept for each strategy we have identified:

- *fillable*: fill the border with a specific value.
- *mirrorable*: mirror the image as if there was an axial symmetry, with the border being the axis.
- *periodizeable*: repeat the image, as if a modulo size was applied to the coordinates.
- *clampable*: extend the value at the image's border into the extension.
- *extent_with*: used when tiling. It considers the current image as a sub-image of another bigger image and pick the extension values from the parent image.

Those concepts are defined in table C.26 and their behavior is described in appendix C.3.3. All those concepts allow us to introduce the final refined image concepts: *WithExtensionImage*, *ConcreteImage* and *ViewImage*. Those two last will be seen in detail in the next chapter 4. Those concepts are defined in table C.28. Their behavior is described in appendix C.3.4. It is now possible to write the following code:

```
template <class I, class SE>
my_local_algorithm(I input, SE se) {
  // if the extension is large enough to function with the passed structuring element
  if(input.extension().fit(se)) {
    // ...
  }
}
```

We show how those three concepts (structuring elements, neighborhood and extension) interact with each other in the diagram shown in fig. 3.16.

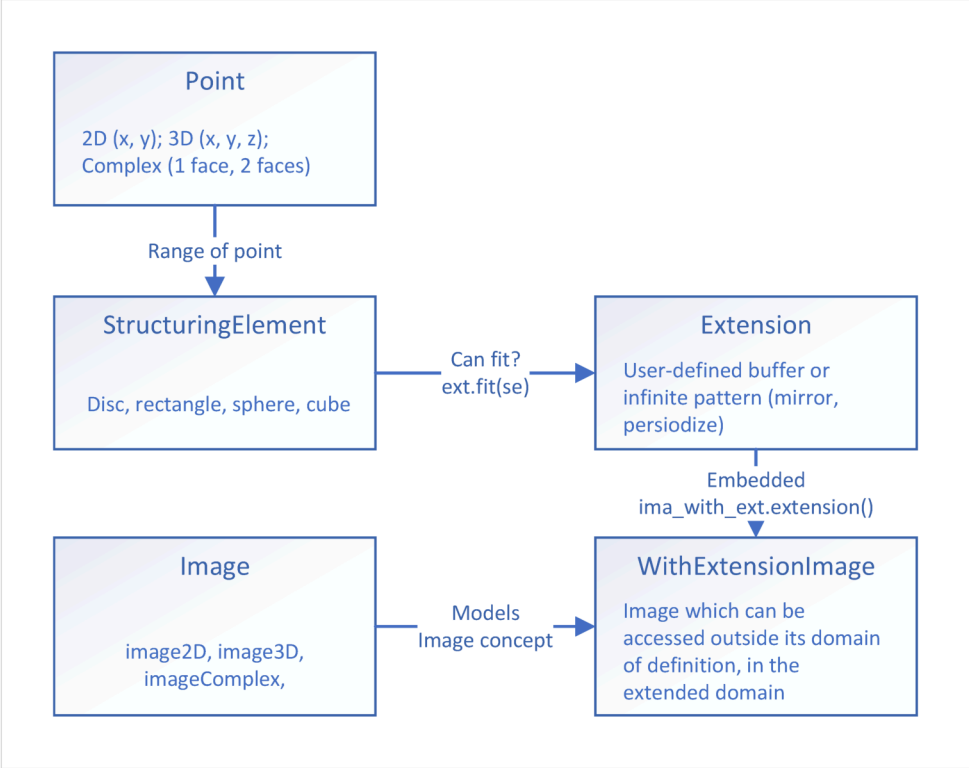


Figure 3.16: Structuring element and Extension concepts.

Finally, we introduce a helper concept to centralize the detection of the "writability" of an image. Indeed, we do not want the user to have to use the writable counterpart of each concept for each and every case. That is why we introduce this final concept, *OutputImage* in appendix C.3.5, that will tell whether the values of an image are mutable.

The correct way to use it is:

```
template <class Img>
requires RawImage<Img> && OutputImage<Img>
void my_algorithm(Img img) {
    // write data in img ...
  }
```

3.5 Summary

In this chapter, we present that concepts are not designed after data structures but after algorithms. Indeed, a concept consists in extracting a consistent behavioral pattern from several pieces of code (algorithms) and name it to give it a meaning. Through a simple but concrete example, we present in a didactic way to extract concepts from an image processing algorithm (gamma correction).

This chapter then proceeds to explain how, in theory, image types are related to each other. We present the set of different image types families and how algorithms exist in those sets, which introduce the notion of *version* of an algorithm. An algorithm have different *versions* for each image types family it supports. We distinguish it from an algorithm *specialization*, the latter being the ability to leverage a property to make an optimization and increase performances.

This chapter then proceeds to describe the notion of algorithm canvas which is the result flowing from the taxonomy of image processing algorithms. Indeed, there are three main algorithm

families: the pixel-wise algorithms (binary threshold), the local algorithms (dilation) and the global algorithms (Chamfer distance transform). We focus primarily on local algorithms and how they can all be written with the same canvas of code. Indeed, for instance, the only difference between a dilation and an erosion is the supremum operator (max vs. min). We then discuss avenues to leverage these canvas to possibly solve heterogeneous computing issues.

Finally, this chapter introduces our first main contribution: a complete taxonomy related to the image processing area. We first introduce fundamental concepts such as *point*, *pixel*, *domain* and *image*. We then motivate and introduce advanced concepts related to images and the different way to access data (forward, backward traversing, indexing, direct access to underlying buffer, ...). In the end, we introduce the concepts related to orbiting notions such as *structuring element*, *neighborhood* and *extension* (border management) which are necessary to be able to work with local algorithms.

The next chapter will make use of the presented concepts to introduce the second main contribution of this thesis: the *image views*.

Chapter 4

Image views

This concept of views is not new [24] and naturally appeared in Image processing with Milena under the name of *morpher* [80, 95]. It was always useful to be able to project an image through a prism that could extract specific information about it without the need to copy the underlying data buffer. In modern days, the language C++ (20) also introduces this mechanism with the ranges [184] facilities for *non-owning collections*. It is named *views* and allows the user to access the content of a container (vector, map) through a prism. In Pylene, we decided to align the naming system after what was decided in C++20 in order not to confuse the user. This way, a `transform` view in image processing will do the same thing on an image that the transform view in the standard range library does on a container. *Views* feature the following properties: *cheap to copy*, *non-owner* (does not *own* any data buffer), *lazy evaluation* (accessing the value of a pixel may require computations) and *composition*. When chained, the compiler builds a *tree of expressions* (or *expression template* as used in many scientific computing libraries such as Eigen [84]), thus it knows at compile-time the type of the composition and ensures a 0-overhead at evaluation. We will first motivate the usage of *views* in image processing. We will then present the main views used in image processing. Then will be discussed how image views differ from the one used in C++'s ranges and their main properties (especially how they keep/discard the properties from the parent image) through a concrete example: the management of border and extension policies. Finally, we will discuss the limitations of such a design.

4.1 The Genesis of a new abstraction layers: Views

In image processing an algorithm is naively written by taking one or several inputs' data (among which is the input image(s)), by performing work on this input data and then by returning the resulting data (or an error). Let us take for example the alpha-blending example which can be implemented in naive C++ code as followed:

```
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);
  }
}
```

This code has several flaws. It makes strong hypothesis about the input images: its data buffer contiguity and its shape (2D). Let us suppose that our user now wants to restrict the algorithm to a specific region inside the image. The maintainer would have then to provide an overload of the algorithm with one additional input argument corresponding to the region of interest. Let us suppose that the user now wants to support manipulate 3D images. The maintainer would now have to provide two additional overloads with an additional stride argument (one for the

base algorithm, one for the region of interest-restricted algorithm). Let us now suppose that the user only wants to manipulate the red color channel. Now the maintainer must support and additional overloads of the algorithm for each channel and/or color type. The complexity increases manyfold for each kind of customization points the maintainer wants to offer to the user. Of course, it is possible to prevent code duplication through clever usage of computer engineering techniques (code factorization etc.) but the complexity would still leak through the API anyway. That is way the other solution is to make the user able to perform those restriction upstream from the algorithm transparently so that the downstream algorithm is easy to write, understand and maintain. In order to achieve this, we need to raise the abstraction level around images by one layer so that we can work at the image level. The alpha-blending algorithm would then be written as shown in fig. 4.1.

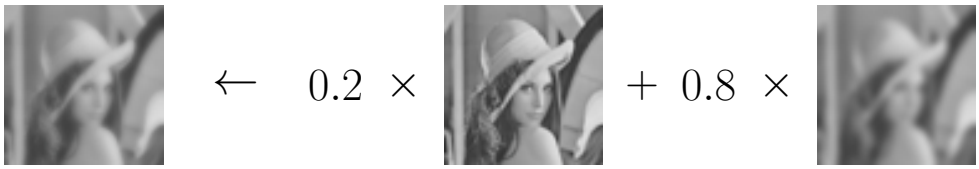
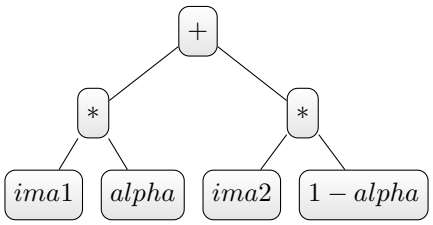


Figure 4.1: Alpha-blending algorithm written at image level.

This way to express an algorithm is achieved by introducing *views* to image processing. An image now is a view and can be restricted/projected/manipulated however the user need before feeding it to an algorithm. Even the whole alpha blending algorithm can be rewritten in terms of views entirely, as shown in fig. 4.2.




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

Figure 4.2: Alpha-blending, generic implementation with *views*, expression tree.

Being able to perform powerful manipulation on images before feeding them to algorithms completely nullify the initial problem of having several overloads of the same algorithm while maintaining and documenting all the associated optional arguments. Indeed, in order to perform the alpha-blending transformation on the base input image, all that the user must do is:

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

If the user wants to restrict the region to be blended, or the color channel to work on, he just has to write the following modification:

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

The restriction is done upstream from the algorithm and propagated downstream without increasing the code complexity. This way, view greatly increase what the user can do by writing less code. The authors explain in detail in [179] how to turn a pixel-wise image processing algorithm into a new image type which is an image view.

4.2 Views for image processing

There are four fundamental kinds of views, inspired by the functional programming paradigm: `transform(input, f)` applies the transformation f on each pixel of the image *input*, `filter(input, pred)` keeps the pixels of *input* that satisfy the predicate *pred*, `clip(input, domain)` keeps the pixels of *input* that are in the *domain*, and `zip(`$input_1$`, `$input_2$`, ..., `$input_n$`)` allows to pack several pixels of several images to iterate on them all at the same time. From those four fundamentals come out more very useful views such as `cast<T>(input)` or `mask(input, msk)` that are more specific to the image processing area.

In Pylena, the practitioner can use a large array of views. Those views come into different form and allow the practitioner to seamlessly use arithmetic or logic operators on images like he would when using expression template. We separate the available views in two main families: the views that perform a restriction of the domain (clip, filter) and the views that transform the values (transform, zip).

4.2.1 Domain-restricting views

The filter view It is also a fundamental view which consists in keeping only the values that satisfy a predicate. This is very useful when working with thresholds as shown in the following code:

```
auto my_threshold = 145;
auto inferior_to [my_threshold](uint8_t val) { return val <= my_threshold; };
auto superiorstrict_to = [](uint8_t val) { return not inferior_to(val); };
mln::image2d<uint8_t> ima_grayscale = /* ... */;
auto ima_inferior = mln::view::filter(ima_grayscale, inferior_to);
auto ima_superiorstrict = mln::view::filter(ima_grayscale, superiorstrict_to);
mln::fill(ima_inferior, 0u8);
mln::fill(ima_superiorstrict, 255u8);
```

This code shows a way to binarize `ima_grayscale` with a custom threshold using the filter view. It is important to note that the resulting filtered image has its domain of definition changed. And the new domain of definition will most likely not be in a regular usual shape (such as a 2D rectangle). This implies that the usage of this view inside certain algorithms may be limited.

The clip view It is a convenient way to extract a sub-image from a base image. This view essentially redefine the domain of definition to restrict it into a smaller one. It does not change anything else which means it proxies every access to the image. For instance, we make use of this view to easily subdivide a 2D-image into 4 tiles as shown in the code below:

```
mln::image2d<mln::rgb8> large_image = /* ... */;
point2d shape = large_image.domain().shape();
auto middle_pnt = point2d{shape.x() / 2, shape.y() / 2};
auto tl = large_image.domain().tl(); // top-left point
auto br = large_image.domain().br(); // bottom-right point
auto four_tiles = std::tuple{
  mln::view::clip(ima, mln::box2d{tl, middle_pnt}),   // top-left tile
  mln::view::clip(ima, mln::box2d{                    // top-right tile
    point2d{middle_pnt.x(), tl.y()},
    point2d{br.x(), middle_pnt.y()}
    }),
  mln::view::clip(ima, mln::box2d{                    // bottom-left tile
    point2d{tl.x(), middle_point.y()},
    point2d{middle_pnt.x(), br.y()}
  }),
  mln::view::clip(ima, mln::box2d{middle_pnt, br})   // bottom-right tile
};
```

The mask view It is very image-processing oriented as it allows the practitioner to provide a boolean image the same size as the original image to select only the pixels whose corresponding value in the mask is true. Its usage is shown in the following code:

```
mln::image2d<mln::rgb8> ima = /* ... */;
auto mask = ima > 127;
mln::fill(mln::view::mask(ima, mask), 255);
```

This code set all the values that are superior to 127 to the max value 255. It shows that it can both be used with read and write access.

4.2.2 Value-transforming views

The transform view It is the most important view of all. It consists in applying a function to each image's pixel. For instance, writing the grayscale algorithm with a transform view is as simple as the following code:

```
auto grayscale_transform = [](mln::rgb8 val) -> uint8_t {
  return 0.2126 * v[0]    // red
         + 0.7152 * v[1]  // green
         + 0.0722 * v[2]; // blue
};
mln::image2d<mln::rgb8> ima_rgb = /* ... */;
mln::image2d<uint8_t>   ima_grayscale = mln::view::transform(ima_rgb, grayscale_transform);
```

There is no loop in this code, just the pixel-wise transformation function. Furthermore, the code will not compute the resulting image. The computation will happen on-the-fly each time a value from `ima_grayscale` is requested. This view allows the practitioner to quickly write and adapt any pixel-wise algorithm he needs for his more complex calculation, efficiently.

The zip view It is one of the most useful view and allow the practitioner to iterate over a set of image at the same time. The basic use-case consists in iterating over a set of input image and the output image to be able to consistently assign output values to a resulting computation from input values. Its usage is shown in the following code:

```
mln::image2d<uint8_t> input = /* ... */;
mln::image2d<uint8_t> output{input.domain()};
auto zipped_ima = mln::view::zip(input, output);
for (auto&& [v_in, v_out] : zipped_ima.values())
  v_out = v_in < 145 ? 0 : 255; // binarisation
```

This code is another example of how to compute a binary threshold on an image.

The channel/RGB views It is a projector to access a specific color channel of an image. There exists image with many more channels than just the standard red/green/blue ones, from the astrophysics or medical area for instances. This view is a tool to restrict an image and only access a specific channel. Its usage is shown in the following code:

```
mln::image2d<mln::rgb8> ima = /* ... */;
mln::copy(mln::view::red(ima), mln::view::green(ima));
```

This code copies the red component into the green component. It shows that the view can be used in both read and write access. Another more generic view exists; `mln::view::channel(ima, k)`, that access the `k`-th channel in `ima`.

The cast views It is a way to convert an image's underlying type to another type, by performing a cast. As this does not modify the underlying value in itself, the write access cannot be granted. This view can be used as shown in the following code:

```
mln::image2d<double> ima = /* ... */;
mln::image2d<uint8_t> ima_8bits = mln::view::cast<uint8_t>(ima);
```

The arithmetical operators $+, -, *, /, \%$ are implemented in the form of transformation views that operate point-wise between two images whose size is identical. For instance, writing the following code:

```
mln::image2d<uint8_t> ima1 = /* ... */;
mln::image2d<uint8_t> ima2 = /* ... */;
auto ret = ima1 + ima2;
```

Is equivalent to writing the following code:

```
auto ret = mln::view::transform(ima1, ima2, [](auto v1, auto v2){ return v1 + v2; });
```

It is important to note that the - unary operator is also supported: `-ima1`.

The logical operators $<, <=, ==, !=, >, >=$ are implemented in the same way that arithmetical operators are. Both unary and binary operators are expressed as transform, and writing the following code:

```
auto ret = !ima1 && ima2;
```

Is equivalent to writing the following code:

```
auto tmp = mln::view::transform(ima1, [](auto v){ return !v; });
auto ret = mln::view::transform(tmp, ima2, [](auto v1, auto v2){ return v1 && v2; });
```

It is far more expressive and more comprehensible by the practitioner. Also, a new facility is introduced to express the logic behind a ternary expression (if C then A else B): the operator $ifelse(C, A, B)$. The rationale is to be able to swap between values depending on a boolean mask. This way, a mathematical morphology algorithm such as *hit or miss* can be implemented in the following simple manner:

```
mln::image2d<uint8_t> ima = /* ... */;
auto ero = erode(ima);
auto dil = dilate(ima);
uint8_t zero = 0;
auto ret = mln::view::ifelse(dil < ero, ero - dil, zero);
```

Everything is taken care of and the practitioner just has to write his algorithm.

The mathematical operators They are implemented in the form of views that operates point-wise. The supported mathematical operators are the following: abs, pow, sqr, cbrt, sqrt, sum, prod, min, max, dot, cross, l0norm, l1norm, l2norm, l2norm_sqr, linfnorm, lpnorm, l0dist, l1dist, l2dist, l2dist_sqr, linfdist, and lpdist. Calling an operator onto an image is equivalent to calling a transform view on each value of this image:

```
auto ima = /* ... */;
auto ret = view::maths::abs(ima);
```

Is equivalent to calling:

```
auto ima = /* ... */;
auto ret = view::transform(ima, [](auto v){ return std::abs(v); });
```

4.3 View properties

Views feature interesting properties, especially how they keep/discard the properties of the concrete image they are based on. However, before talking about those properties, it is important to draw the line and point the main differences between the C++20 ranges views and our image views.

4.3.1 Differences between C++20 ranges views and image views

C++20 ranges views are a new abstraction layer introduced on top of the already existing iterators. This means that a view is created from an existing container from its iterators. There are also special views such as `std::views::iota` that are able to generate an infinite sequence of number. Those last are the generator views and are designed to be used the same way as a container, except that they do not own any data. In C++20 the views are mainly constructed from a container such as `std::vector`, `std::map` or `std::list`. For instance, the way to create a view featuring all the elements of a container is shown in the following code:

```
auto vec = std::vector { /* ... */ };
auto vec_vw = std::views::all(vec);
```

This induces issues regarding dangling references when passing temporary views or when the container owning the data expires. To summarize the model of ranges in C++20, they are a new abstraction layer much more friendly and powerful than iterators and can construct non-owning, cheap-to-copy views from an owning container.

4.3.2 Data ownership

The concept of *View* brought to us a fundamental issue when dealing with images: "*What is an image?*". More precisely: should an image always be the owner of its data buffer? Should we have a shared ownership of the data buffer between all the images using it? Then what happens when the data changes? The issue about the semantic of an image is crucial but also very similar to the issue there is to differentiate a *container* (such as `std::vector`, that is to say the data buffer) and a *view*, as seen in section 4.3.1.

From here we have considered two approaches. The first one is to have *shared ownership* of the data buffer for the image and its derived views. However, this does not allow the differentiation between an already computed image and a lazy image. To be able to make this differentiation is crucial in an *Image Processing library* as we want to make the most out of the data we already have, and we do not want to compute data we do not need. Also, we cannot distinguish when the *copyability* property is required. This is the main reason why we did not adopt this approach.

The second one is to make the differentiation between a *concrete image* which owns the data (like the standard containers) and the *views* that are lightweight cheap-to-copy objects. However, this would imply we have to distinguish both image families when writing algorithms, and we do not want that. Indeed, that would defeat the purpose of genericity.

This is why we have chosen to take a path where we mix both approaches at the same time. We assert that all image are cheap-to-copy (including views), even the concrete images. The concrete image will have a shared ownership semantic related to its data buffer and will remain cheap-to-copy. It will also behave the same way a view behaves. That is why, with our semantic, all images are views. Ultimately, there is still a way to distinguish a view from a concrete image, if needed, and we introduce two new concepts `ViewImage` and `ConcreteImage` for this purpose:

```
template <typename I>
concept ConcreteImage =
  Image<I> &&
  std::concepts::semiregular<I> &&  // A concrete image is default constructible
  not image_view_v<I>;

template <typename I>
concept ViewImage =
  Image<I> &&
  image_view_v<I>;
```

Having images as views is a very important property as it simplify greatly the reasoning when performance is needed. This allows the user to pass his images everywhere without worrying about dangling data buffer expiring around the corner. It also enables us to have a library design similar to the C++'s standard library which the user is familiar with and, why not, have

standard algorithm and standard view work on our image types. All of these are the main reason why we decided to adopt this design. However, unlike the standard library, we are not required to work with iterator due to backward compatibility.

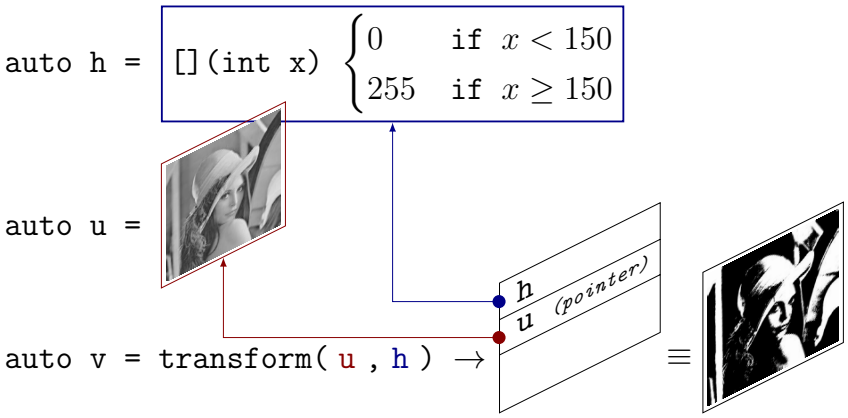


Figure 4.3: An image *view* performing a thresholding.

In our design, all images are lightweight (cheap-to-copy) objects with shared ownership over the data. A view image is a non-owning image that only stores pointers, as shown in fig. 4.3. A concrete image stores the data. The only difference between a view and a concrete image is given by a trait `image_view` which will check the `view` property of the given image to tell whether it is a concrete owning image type or not. Compared to the C++20 ranges views model where no mechanism prevent errors resulting from dangling references and confusing ownership of data. It is especially adapted to image processing as the user generally wants to avoid deep copy of its data. Indeed, when the user wants a deep-copy (clone), he wants to do it explicitly.

This design induces one major property which is the *lazy-evaluation* of the views.

4.3.3 Lazy evaluation, composability and chaining

The major key point of views is the lazy evaluation. When a concrete image is piped through a view, no computation is done. The computation happens when the practitioner requests a value by doing $val = V(p)$. The implications are multiples: an image can be piped into several computation-heavy views, some of which can be discarded later on, and it will not impact the performance. Also, when processing large images, applying a transformation on a part of the image (such as clipping or filtering) is as simple as restricting the domain with a view and applying the transformation to this resulting sub-image.

Lazy-evaluation combined with the view *chaining* allows the user to write clear and very efficient code whose evaluation is delayed till very last moment as shown in fig. 4.4 (see [142] for additional examples). Neither memory allocation nor computation are performed; the image i has just recorded all the operations required to compute its values.

The tree of type resulting from this view chaining is illustrated by fig. 4.5. It illustrates how chaining views with each other result in the formation of an abstract tree that records the operations to perform. This model allows building complex computational trees via views while keeping efficient performance at runtime. However, those trees are complex for the compiler to process and can induce substantial compilation time overhead.

4.3.4 Preserving image properties

Views will also try to preserve properties of the original image when they can. That means that views can preserve the ability of the practitioner to, for instance, write into an image. This may be a trivial property to preserve when considering a view that restrict a domain, but when

```
image2d<rgb8>  ima1 = /* ... */;
image2d<uint8_t> ima2 = /* ... */;

// Projection: project the red channel value
auto f = view::transform(ima1, [](auto v) {
  return v.r;
});

// Lazy-evaluation of the element-wise
// minimum
auto g = view::transform(view::zip(f, ima2),
  [](auto value) {
    return std::min(std::get<0>(value),
             std::get<1>(value));
});

// Lazy-Filtering: keep pixels whose value
// is below < 128
auto h = view::filter(g, [] (auto value) {
  return value < 128;
}));

// Lazy-evaluation of a gamma correction
using value_t = typename Image::value_type;
constexpr float gamma = 2.2f;
constexpr auto max_val =
  std::numeric_limits<value_t>::max();
auto i = view::transform(h,
  [gamma_corr = 1 / gamma, max_val] (auto value) {
    return std::pow(value / max_val,
             gamma_corr) * max_val;
});
```

Figure 4.4: Lazy-evaluation and *view* chaining.

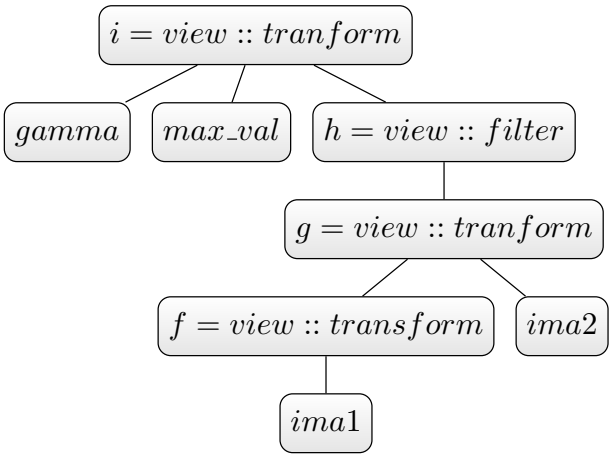


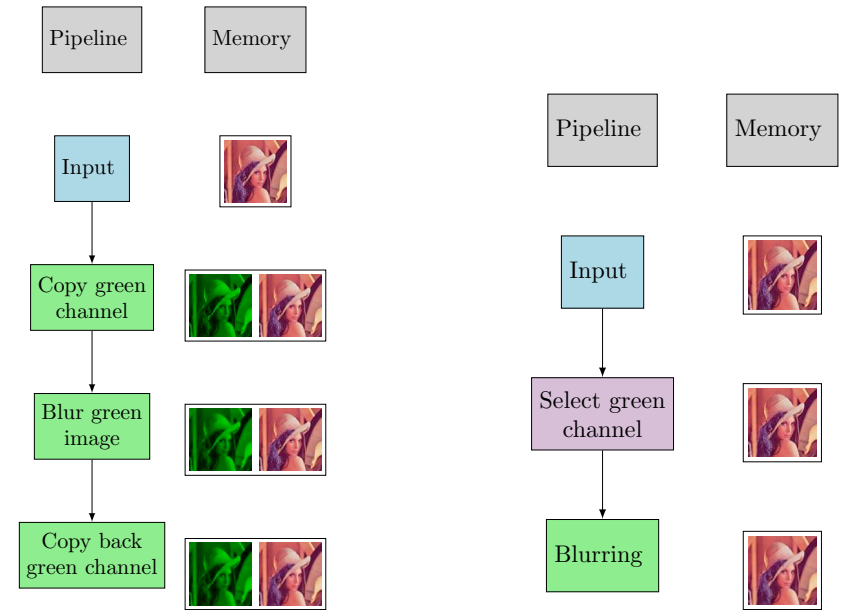
Figure 4.5: Abstract Syntax Tree of the types chained by the code in fig. 4.4

considering a view that transforms the resulting values, it is not. Let us consider the projection $h : (r, g, b) \mapsto g$ that selects the green component of an RGB triplet. When piping the resulting view into, for instance, a blurring algorithm, the computation will be performed in place thanks to the fact we still have the ability to write into the image. A legacy way of obtaining the same result would have been to create a temporary single-channel image to extract the green channel of the original RGB image so that the temporary image could then be blurred. The final step would have then been to copy the values of the temporary blurred image back into the green channel of the original image. The comparison, including the memory used, between the legacy way and the in-place way of doing this computation is shown in fig. 4.6.

On the other hand, when considering the view $g : (r, g, b) \mapsto 0.2126 * r + 0.7152 * g + 0.0722 * b$ that compute the gray level of a color triplet (as shown in fig. 4.7), the ability to write a value into the image cannot be preserved. Indeed, one would need an inverse function that is able to deduce the original color triplet from the gray level to be able to write back into the original image. This operation alone is a whole field of research on its own [133, 59, 50]

Similarly, a view can apply a restriction on an image domain. In fig. 4.8, we show the adaptor `clip(input, roi)` that restricts the image to a non-regular `roi` and `filter(input, predicate)` that restricts the domain based on a predicate. All subsequent operations on those images will only affect the selected pixels. In this case of restriction, the ability to write data back into the original image is preserved through the view.

Views feature many interesting properties that change the way we program an image processing application. To illustrate those features, let us consider the following image processing pipeline: (Start) Load an input RGB-8 2D image (a classical HDR photography) (A) Convert it in



(a) Legacy pipeline with copy

(b) Modern pipeline with view

Figure 4.6: Comparison of a legacy and a modern pipeline using *algorithms* (green) and *views* (purple).

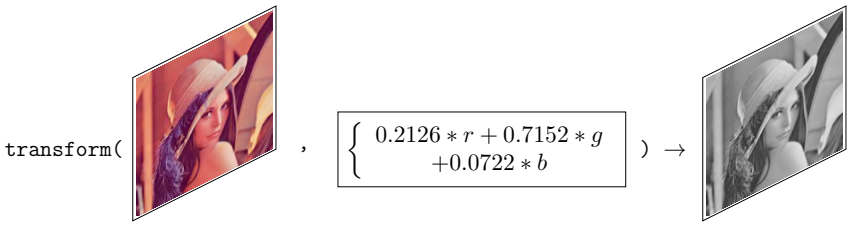


Figure 4.7: Usage of transform view: grayscale.

(a) Clip view

(b) Filter view

Figure 4.8: Clip and filter image adaptors that restrict the image domain by a non-regular ROI and by a predicate that selects only even pixels.

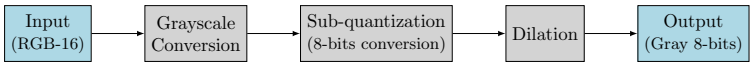


Figure 4.9: Example of a simple image processing pipeline.

grayscale (B) Sub-quantize to 8-bits (C) Perform the grayscale dilation of the image (End) Save the resulting 2D 8-bits grayscale image; as described in fig. 4.9. This pipeline is expressed with two notions. The first notion is composition of algorithms $(A \to B \to C)$ in order to achieve the desired result. The second notion is the composition of views $(Input \to A \to B)$ which overlaps partially with the algorithm part. This express that part of the algorithm is performed lazily only when we perform the last part (C). Those two notions are illustrated by the fig. 4.10.

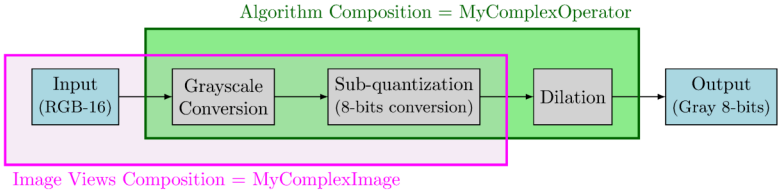


Figure 4.10: Example of a simple image processing pipeline illustrating the difference between the composition of algorithms and image views.

There are six properties one want to keep track when working with views: *forward*, *writable*, *accessible*, *indexable*, *bidirectional* and *raw*. Those properties echo to the concepts seen in section 3.4. An image is *forward* when it can be traversed in a forward way. It is *writable* when the values are mutable. It is *accessible* whenever it allows to access the value associated to a point (i.e. it allows to write the expression $v = ima(p)$). It is *indexable* whenever its values can be accessed through an index localizer (i.e. it allows to write the expression $v = ima[idx]$). Usually, accessing through an index is faster than accessing by a point. It is *bidirectional* when it can be traversed in both a forward and a backward way. Finally, an image is *raw* when its data buffer is contiguous and can directly be accessed with information about strides. The table 4.1 presents all the views and how they preserve the base properties of a concrete image.

Table 4.1: Views: property conservation

View type	Property / Expression	Forward	Bidirectional	Raw	Writable	Accessible	Indexable
Image	`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)`.

Also, we may want to extend the property preservation discussion to other concepts we saw in the previous chapter 3, especially the concepts of structuring element and extension.

4.4 Decorating images to ease border management

When looking at local algorithms, we notice that a long recurring issue is regarding the behavior on the border of the image. There are many ways of dealing with this problem. One is to allocate additional memory for the border and paste values in it. Another is to check the bounds when looping over the neighbors inside the computational window. We can also decorate the image to return a correct lazily computed value when accessing out-of-image-bound value still inside the extension. The point is: all these methods have advantages as well as disadvantages.

Memory allocated border The border width is fixed at the image's creation and cannot be augmented without doing a reallocation. There is also a cost when computing border's values (to fill it) which is proportional to the border's width and to the image's size. On the other hand, the access time of a border value during the algorithm unrolling is as fast as a native access time within the image itself. The last issue remaining would be that the border is not infinite. We cannot process a local algorithm with a structuring element that does not fit in the extension. This method is especially adapted when there is medium structuring elements with a known size which will yield a lot of out-of-image's bound accesses. When speed is required, this method is a de facto standard.

Bound checking Assuming there is no border, and we are not allowed to access out-of-image-bound values, a bound check is required when accessing each values. Another way to do would be to decorate the facility that yields the neighbors of a pixel: do not yield out-of-image-bound pixels. This removes the need to bound check for each pixel's value which is relatively faster. The caveats of this method are that it induces a slight slow down when yielding the pixel's neighbors from the structuring element, and that it is not always viable: some algorithms do need to access values in an extension to produce proper results.

Image decoration The border is infinite, and we make a view of our image to decorate it with the required extension. This is achieved using *views*: the original image is chained into a view that will add the required feature to the image. For instance, let us consider the following image:

```
struct borderless_image {
  // ...
  // NO extension_type subtype
  // NO extension() method
};
```

Attempting to use this image in a local algorithm that works with a structuring element will raise an error, as the structuring element does not fit inside the image when considering the behavior on the borders. However, instead of narrowing the region of interest (i.e. casting out bordering pixels), it is possible to make a view that will return an image for which the behavior at the border is well-defined. Referring to the taxonomy from the previous chapter 3 we remember that we can construct a custom extension type for the sake of an example (as described in section 3.4.3). This example will decorate the image so that the border is always filled with a specific value. The following code shows how we can write such an extension:

```
template <class ValueType>
struct FillExt {
  using support_fill = std::true_type; // Support the fill policy
  bool is_fill_supported() const { return true; }

  using value_type = ValueType;        // Underlying value_type

  // Always fit structuring element of any size
  template <class StructuringElement>
  bool fit(const StructuringElement& se) const { return true; }
```

```
  void fill(ValueType v) { v_ = v; }  // Assign the filled value
  // ...
  // Yield the value for a given point (always return filled value)
  template <class PointType>
  ValueType val(PointType) const { return v_; }
  // ...
  private:
    ValueType v_;
};
```

Now that our extension type is written, we introduce a new image type which will adapt our base `borderless_image` into a view which features out `FillExt` extension type.

```
template <class BaseImage>
struct filled_border_image : image_view_adaptor<BaseImage>{
  // ...
  using extension_type = FillExt;
  extension_type& extention() const { ext_; }
  // ...

  value_type at(point_type pnt) {
    if(!domain().has(pnt)) {
      return extension().val()
    }
  }
  // ... adapt all the methods that can make out-of-bound access and fallback on
  // the extension's value ...
private:
  extension_type ext_;
};
```

Finally, all that is left to do is to write the function that will construct the view from the base image:

```
template <class BaseImage>
auto fill_extension_view(BaseImage ima) {
  // call to the image_view_adaptor ctor
  auto with_fill_ext_ima = filled_border_image<BaseImage>(ima);
  return with_fill_ext_ima; // <-- this is a view
}
```

This simple function enables very powerful usage as illustrated in the code below:

```
  auto ima = image2d<uint8_t>{
    {0, 1, 2},
    {3, 4, 5}
    {6, 7, 8}
  };
  ima.at({1, 1}); // OK, 4
  ima.at({5, 5}); // ERROR, out-of-bound
  // Get a view
  auto ima_with_filled_border = fill_extension_view(ima);
  // Fill border with value 255
  ima_with_filled_border.extension().fill(255);
  ima_with_filled_border.at({5, 5}); // OK, 255
```

For the sake of brevity we have simplified the implementation in our example. In practice the implementation of such a pattern is more complex as there are many strategies to support, the interfaces of the extension may be different, the decoration of the image type may not be enough, notably for the *none* strategy where it is required to decorate the structuring element.

At the end, this method has the advantage to *always work*. Given any structuring element of any size, any algorithm will work. The disadvantage is that we need to check for out-of-bound access at the image level, and lazily compute the value in case of out-of-image-bound access. The slowness induced is not negligible and should be weighted carefully.

It is important to note the very close relation between an image's domain (to perform out-of-bound checks), the structuring element (notably its size) and the extension (its width). A user may require, for a specific set of those three elements, to decorate the image, and/or the

structuring element and/or to perform computation and/or reallocation. To resolve this issue, we decided to provide the user with a new facility: the *border manager* whose job is to prepare a suitable pair (image and structuring element) given a set of configuration wanted by the user.

We designed the configuration to be constructed from a given set of a policy and a method. We currently offer two policies: native and auto.

- Native: if the border is large enough: forward the image as-is to the algorithm to allow the fastest access possible. Otherwise, the border manager fails and halt the program.
- Auto: if the border is large enough: forward the image as-is to the algorithm to allow the fastest access possible. Otherwise, decorate the image with a view whose extension will emulate what is required by the algorithm with the given structuring element.

We also provide seven different methods to fill up our extension with the wanted values. It is important to note that not all the methods are available for both policies. The policies are: none, fill, mirror, periodize, clamp, image and user.

The *none* policy enforces a policy where there is no border to use. This method cannot fail as it makes the border vanish. To enforce this method, the border manager decorates the structuring element in a view that checks the domain inclusion of each neighboring point. The *fill* policy enforces that the border is filled with a specific value. The *mirror* policy enforces that the border is filled with a mirrored value from an axial symmetry relative to the image's edges. The *periodize* policy enforces that the border replicate the image, like a mosaic. The *clamp* policy enforces that the border is filled with values expanded from the values at the image's edge. The *image* policy enforces all points out of the current image's domain are to be picked inside another image. A basic use-case is preparing tiles from a larger image. The position of our image can be offset in the image acting as an extension which ease the usage when, for instance, clipping a sub-image. The fig. 4.11 shows how a sub-image (tile) can consider the base image as its border. Finally, the *user* policy assumes the user knows what he is doing and do not touch nor decorate the given image in any way. The fig. 4.11 illustrates all the border methods mentioned.

As a consequence the usage of a local algorithm becomes very simple:

```
// default border width is 3
image2d<int> ima = {{0, 1, 0}, {0, 1, 1}, {0, 1, 0}};
auto disc_se = se::disc{1}; // radius is 1
auto bm = extension::bm::fill(0); // fill border with 0 with policy auto

local_algorithm(ima, disc_se, bm); // will handle the border for you
```

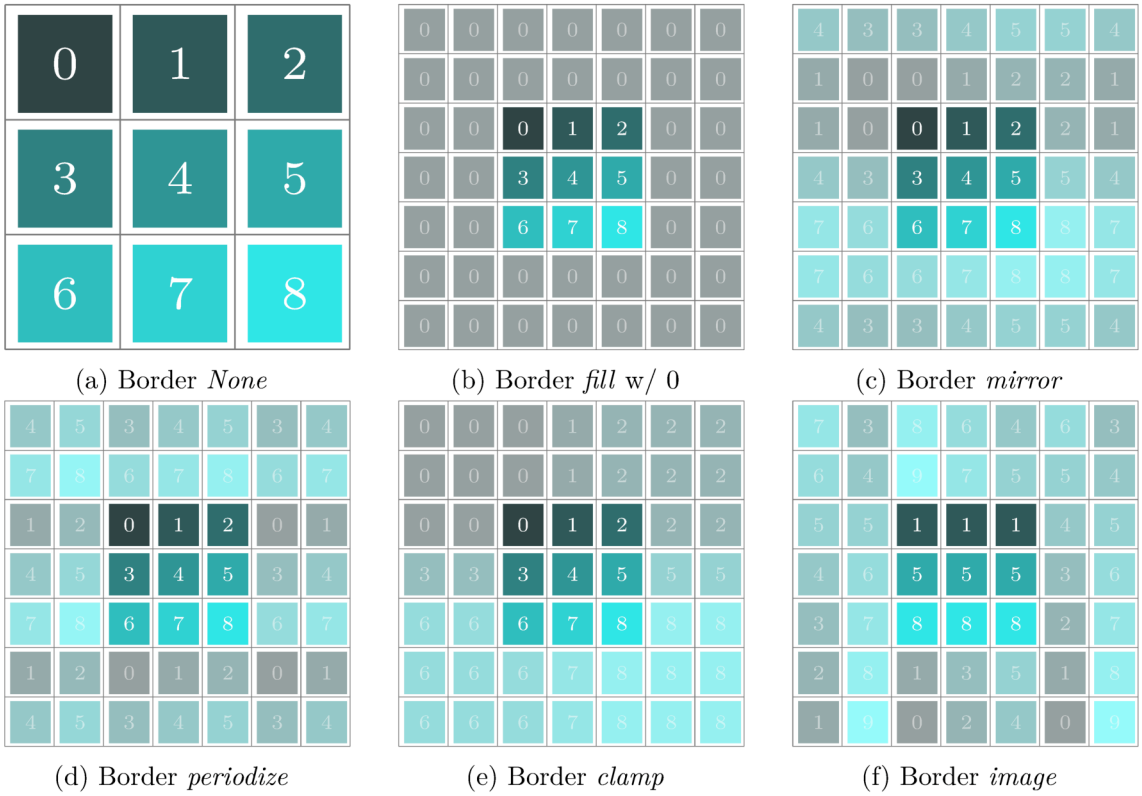
The border manager `bm` is set with the method fill (with value 0) and the policy `auto` (which is the default policy). To use the policy `native`, one would write `extension::bm::native::fill(0)` instead.

In the implementation of the local algorithm, a dispatch is made with the pattern *visitor*, relying on the standard facilities `std::visit` and `std::variant` so that the performance overhead as well as the complexity of use remain minimal. Let us assume we have a local algorithm implemented this way:

```
template <class Ima, class SE>
local_algorithm(Ima ima, SE se)
{
  // assume ima has a large enough border for the given se
  // use ima & se in loop
}
```

We can rewrite it leveraging the border manager facility this way:

```
template <class Ima, class SE, class BM>
local_algorithm(Ima ima, SE se, BM bm)
{
  auto [managed_ima, managed_se] = bm.manage(ima, se);
```



(a) Border *None* (b) Border *fill* w/ 0 (c) Border *mirror*

(d) Border *periodize* (e) Border *clamp* (f) Border *image*

Figure 4.11: Border methods' breakdown.

```
  std::visit([&](auto&& ima_, auto&& se_) {
    // use ima_ and se_ in loop
    }, managed_ima, managed_se);
}
```

The overhead is kept minimal thanks to using `std::variant` and `std::visit` and the algorithm implementer delegates the border management to the border manager. This is made possible thanks to the views. Indeed, under the hood the border manager may pipe the original image into a view that will behave accordingly to the policy chosen by the user. This is transparent from both practitioner and maintainer points of views.

4.5 Views limitations

Views can be of tremendous use in our area however it relies on metaprogramming techniques which are infamous for greatly increasing the compilation time of source code. Also, when one starts to chain views a lot, combining different image type (via *zip* for instance), combined with the overhead induced via the border manager using `std::variant`, the compilation time can really become an issue. Indeed, C++ developers tend to minimize the cost of compilation time because once the program is compiled, the binaries can be distributed and are really fast to execute. However, we are not exactly in that case as our library is generic. That means we distribute source code to our user and our user compiles it when prototyping their program. This is an issue every library developer faces: distributing heavily templated source code as a library can be a deal-breaker. In the industry, it was even to the point that people refused to use boost in their code line. The boost maintainers had to modularize their library, so that users were able to cherry-pick the parts they needed without pulling half of the library which was a disaster for the compilation time of their project.

The ranges for C++20's standard library and its views face the same issue. It was not rare for someone to need 90sec to compile the calendar toy example of the library which just contains

code that displays a given month in the classic printed format (day number-of-month correctly displayed in column corresponding to the day of week label). This massive compile-time is due to early compiler implementation needing massive RAM usage for template type and having to do memory swap on disk when the computer running the compilation was out of RAM. Nowadays, compilers have optimized the whole process, but the combinatorial explosion behind the types can still be an issue. Introducing complex view code in a program that is compiled often may not be a good idea. However, a program that is rarely compiled but is run a great number of time may take advantages of all the optimizations the compiler can do to remain very efficient.

4.5.1 Image traversing with ranges

Lastly, views usage should be measured when used at critical points. We learned from experience that one simple change can make the compiler miss optimization opportunities which can greatly impact the resulting performance. Let us illustrate our remark with a concrete example: image traversing. In a previous version of our library, we used macro for image traversing. `mln_concrete`, `mln_piter`, `mln_qiter`, `for_all` and `mln_value` are all macros aiming at hiding the underlying complexity. Our goal were to replace those macros with existing C++ core language code to improve the user experience as well as ease the maintenance, contribution and further improvement of the library. To do so, we based our image traversing on `std::ranges`. Let us take the old implementation (from Milena [85] we had of our dilation algorithm as an example:

```
template<class I, class 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;
}
```

This code features the macro mentioned above and, while being explicit, may be quite obscure with regard to its internals for a non-initiated user. However, it uses an in-house iterator on points to traverse the image under the hood. This in-house iterator prevents the compiler to make crucial optimizations, such as vectorization, which hinders greatly the performances. Rewriting the algorithm using ranges in modern code results in the following code:

```
template<class I, , class SE>
auto dilate(I input, const SE& se)
{
  auto output = input.concretize(); // clone image
  for(auto [in_px, out_px] : view::zip(f.pixels(), g.pixels()))
  {
    out_px.val() = out_px.val();
    for(auto nhx : se(in_px))
      out_pix.val() = std::max(nhx.val(), out_px.val());
  }
  return output;
}
```

This code use the *zip* view to iterate over two images (the input image and the resulting output image) simultaneously. This is native code, and it should, in theory, at least be as efficient as the old version of the code (with macros) as it allows compiler optimizations such as vectorization or

inner loops unrolling. But through benchmarking, we have learned that this solution does not mix well [28] with the multidimensional nature of images. The issue originates from the fact that we have no way to explicitly say in the code that the multidimensional range is made of chunk of contiguous rows of memory. Indeed, for each element we have to compute an index originating from potentially N dimensions. This disables critical optimizations such as vectorization, which was a disaster performance-wise. We solved this problem by augmenting range-v3's ranges with our own multidimensional ranges. Indeed, we only need to have contiguity on the last dimension to provide the compiler code it can optimize. Which means that each for-loop that traverses the whole n-dimensional image can be transformed into a double for-loop whose inner loop is guaranteed to be a contiguous row. This way we have now an outer range as well as an inner range, as illustrated in fig. 4.12.

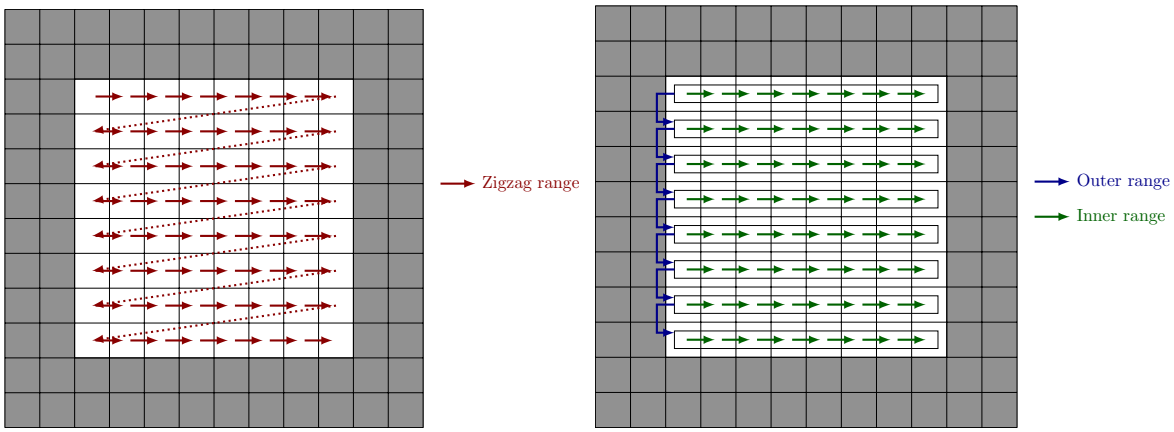


Figure 4.12: Range-v3's ranges (a) vs. multidimensional ranges (b).

Thanks to this new design we can now rewrite our algorithm with a double for-loop for the image traversing. Hopefully it stays really similar to what one would be used to when working with the classical two-dimensional image. As an example, we can rewrite the dilation algorithm this way:

```
template<class I, class SE>
auto dilate(I input, const SE& se)
{
  auto output = input.concretize(); // clone image
  // this line is needed to avoid dangling reference
  auto zipped_pixels = view::zip(input.pixels(), output.pixels());
  for(auto&& row : ranges::rows(zipped_pixels)) // unroll the contiguous segments
    for(auto [in_px, out_px] : row)             // optimized traversing of the segment
    {
      out_px.val() = out_px.val();
      for(auto nhx : se(in_px))
        out_pix.val() = std::max(in_px.val(), out_px.val());
    }
  return output;
}
```

The highlight of this code is the usage a new tool: `ranges::rows` to bring out an inner range (contiguous) from the multidimensional outer range.

4.5.2 Performance discussion

In order to have a relevant discussion on performance, we decided to implement a real world image processing pipeline: the background subtraction. It is used to detect changes in image sequences [138]. It is mainly used when regions of interest are foreground objects. The pipeline

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

Table 4.2: Benchmarks of the pipeline fig. 4.13 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.

components include: subtraction, Gaussian filtering, threshold, erode and dilate, as shown in fig. 4.13.

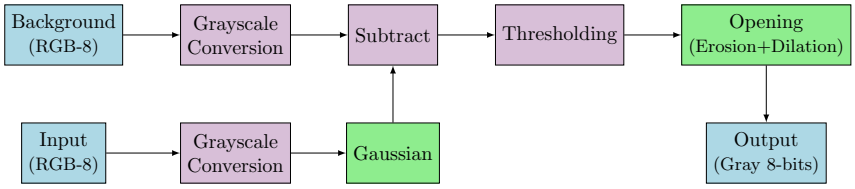


Figure 4.13: Background subtraction pipeline using algorithms and views.

The first thing that we notice is that the implementation of the pipeline using views is transcribed very explicitly in the code, as shown in fig. 4.14. There is a direct correspondence between the graphic pipeline and the code.

For our benchmark, we have decided to run the algorithm on an original set of image to detect a changing foreground. We have considered 10 data set. We present in fig. 4.15 three of them for the sake of brevity.

We have run benchmarks on this set comparing multiple ways of achieving this result, both using Pylene and OpenCV as well as varying the size and the shape of the structuring element window. The breakdown of these benchmarks is presented in fig. 4.16. In table 4.2, we benchmark the computation time and the memory usage[1]of these implementations (all single-threaded) with an opening of disc of radius 32 on 10 MPix RGB images (the minimum of many runs is kept).

The results should not be misunderstood. They do not say that OpenCV is faster or slower but shows that implementations all have the same order of processing time, so the comparison makes sense (the algorithms used in our implementation are not the same as those used in OpenCV for blur and dilation/erosion). It allows us to validate experimentally the advantages of views in pipelines. First, we have to be cautious about the real benefit in terms of processing time. Here, most of the time is spent in algorithms that are not eligible for view transformation.

[1]Memory usage is computed with *valgring/massif* as the difference between the memory peak of the run and the memory peak without any computation (just setup and image loading)

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

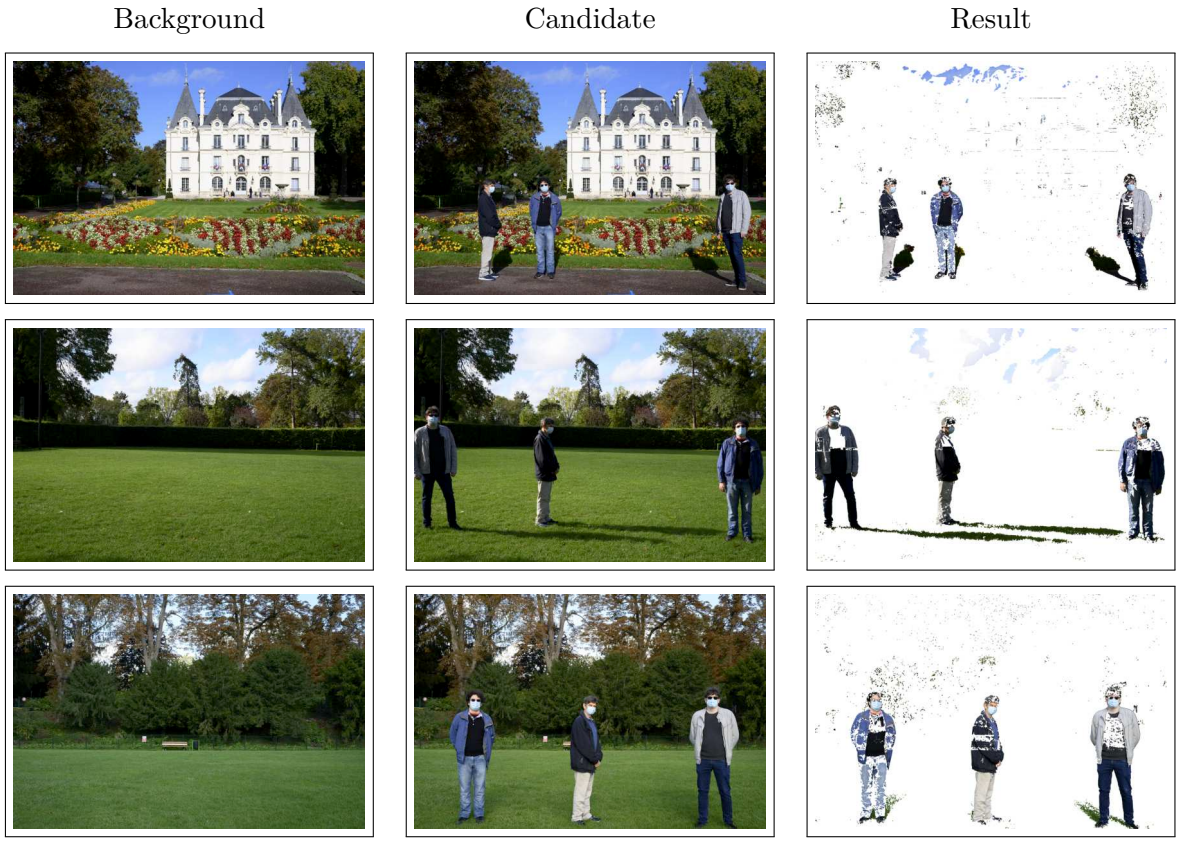


Figure 4.15: Background detection: data set samples.

Thus, depending on the operations of the pipeline, views may not improve processing time. Nevertheless, using views does not degrade performance neither (only 1% in this experiment). It seems to show that using views does not introduce performance penalties and may even be beneficial in lightweight pipelines as the one in fig. 4.9. However, views reduce drastically the memory usage (also seen in fig. 4.6) which is beneficial when developing applications which are memory constrained. From the developer standpoint, it requires only few changes in the code as shown in fig. 4.14 — the implementation of the algorithms remains the same — which is a real advantage for software maintenance.

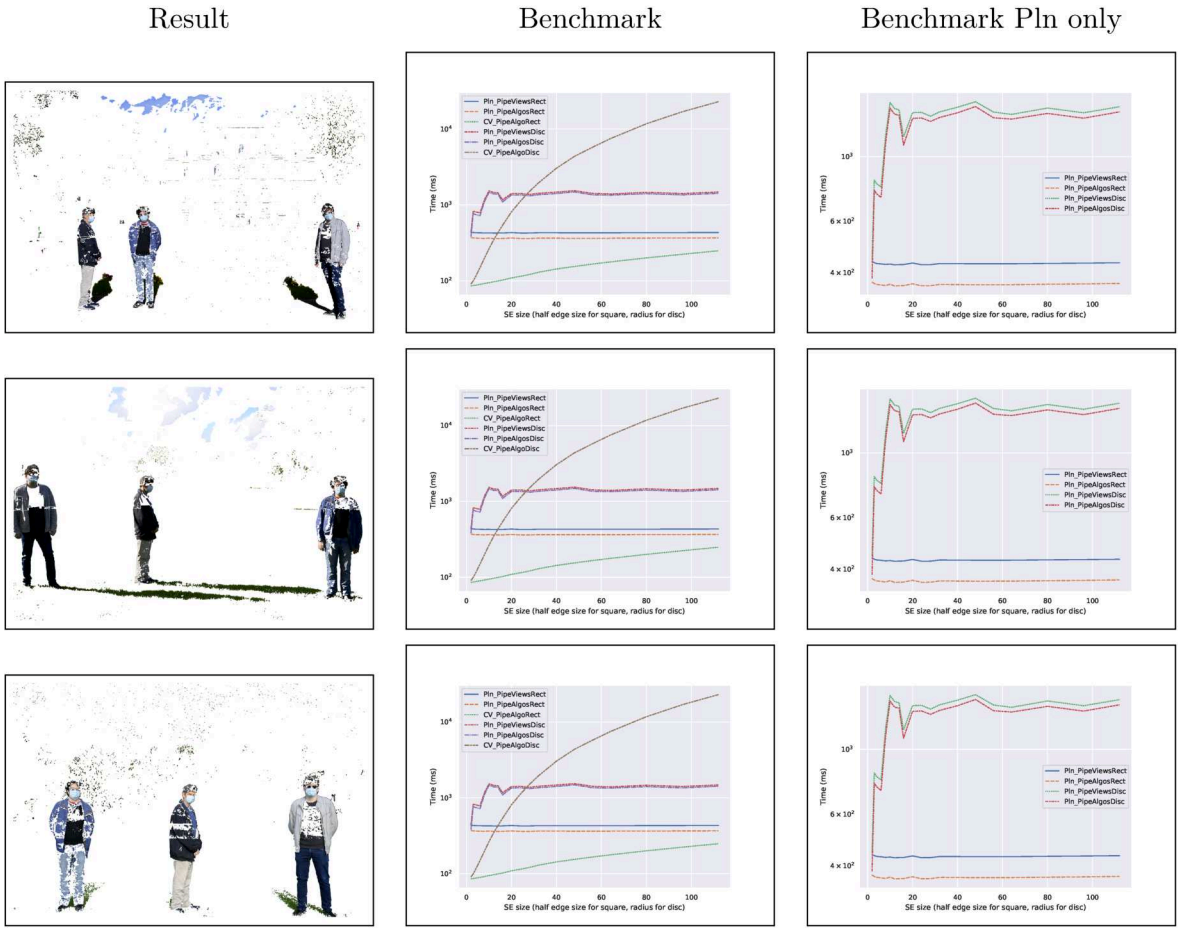


Figure 4.16: Background detection: garden results.

4.6 Summary

Views are composable. One of the most important feature in a pipeline design (generally, in software engineering) is *object composition*. It enables composing simple blocks into complex ones. Those complex blocks can then be managed as if they were still simple blocks. In fig. 4.9, we have 3 simple image processing operators *Image* → *Image* (the grayscale conversion, the sub-quantization and the dilation). As shown in fig. 4.10, algorithm composition would consider these 3 simple operators as a single complex operator *Image* → *Image* that could then be used in another even more complex processing pipeline. Just like algorithms, image views are composable, e.g. a view of the view of an image remains an image. In fig. 4.10, we compose the input image with a grayscale transform view and a sub-quantization view, which is then feed to the dilation algorithm.

Views improve usability. The code to compose images in fig. 4.10 is almost as simple as:

```
auto input = imread(...);
auto A = transform(input, [](rgb16 x) -> float {
    return (x.r + x.g + x.b) / 3.f;
  };
);
auto MyComplexImage = transform(A, [](float x) -> uint8_t {
    return (x / 256 + .5f);
  };
);
```

People familiar with functional programming may notice similarities with these languages where *transform* (*map*) and *filter* are sequence operators. Views use the functional paradigm and are created by functions that take a function as argument: the operator or the predicate to apply for each pixel; we do not iterate by hand on each pixel.

Views improve re-usability. The code snippets above are simple but not very re-usable. However, following the functional programming paradigm, it is quite easy to define new views, because some image adaptors can be considered as *high-order functions* for which we can bind some parameters, as one would do with the curry technique [17]. In fig. 4.17, we show how the primitive *transform* can be used to create a view summing two images and a view operator performing the grayscale conversion as well as the sub-quantization which can be reused afterwards[2].

Views for lazy computing. Because the operation is recorded within the image view, this new image type allows fundamental image types to be mixed with algorithms. In fig. 4.17, the creation of views does not involve any computation in itself but rather delays the computation until the expression `v(p)` is invoked. Because views can be composed, the evaluation can be delayed quite far. Image adaptors are *template expressions* [20, 40] as they record the *expression* used to generate the image as a template parameter. A view actually represents an expression tree (fig. 4.2).

Views for performance. With a classical design, each operation of the pipeline is implemented on "its own". Each operation requires memory to be allocated for the output image and also, each operation requires that the image is fully traversed. This design is simple, flexible, composable, but is not memory efficient nor computation efficient. With the lazy evaluation approach, the image is traversed only once (when the dilation is applied) which has two benefits. First, there are no intermediate images, which is very memory efficient. Second, traversing the image is faster

[2]These functions could have been written in a more generic way for more re-usability, but this is not the purpose here.

```
auto operator+(Image A, Image B) {
  return transform(A, B, std::plus<>());
}
auto togray = [](Image A) {
  return transform(A, [](auto x) {
      return (x.r + x.g + x.b) / 3.f;
    };
  )
};
auto subquantize16to8b = [](Image A) {
  return transform(A, [](float x) {
      return uint8_t(x / 256 +.5f);
    }
  );
};

auto input = imread(...);
auto MyComplexImage = subquantize16to8b(togray(A));
```

Figure 4.17: Using high-order primitive views to create custom view operators.

thanks to a better memory cache usage, and performs an optimal selective traversal. Indeed, in our example (fig. 6), processing a RGB16 pixel from the dilation algorithm directly converts it in grayscale, then sub-quantize it to 8-bits, and finally makes it available for the dilation algorithm. It acts *as if* we were writing an optimal operator that would combine all these operations. This approach is somewhat related to the kernel-fusing operations available in some HPC specifications [150] but views-fusion is optimized by the C++ compiler only [139]. The selective aspect intervenes when a region of interest is selected at one point in the processing pipeline. Indeed, the entirety of the pipeline is then executed only on the region of interest, even if this selection happens only at the very end of the processing pipeline.

Views for productivity. All point-wise image processing algorithms can (and should) be rewritten intuitively by using a one-liner view. The *transform* views is the key enabling that point. This implies that there exists a new abstraction level available to the practitioner when prototyping their algorithm. The time spent implementing features is reduced, thus the feedback-loop time is reduced too. This naturally brings productivity gain for the practitioner.

Part III

Bringing Generic programming to the dynamic world

Chapter 5

A bridge between the static world and the dynamic world

In the programming world, there are three main families of programming language [38]. There are (i) *compiled* programming languages, such as C, C++, Rust or Go, (ii) *interpreted* programming languages, such as Python, PHP or Javascript, and (iii) hybrid programming languages, such as Java or C#. The latter have a fast compilation pass that compiles the source code into an intermediate bytecode. Then, this bytecode is interpreted via an interpreter on the host (runner) machine.

5.1 Introducing the static and dynamic bridge

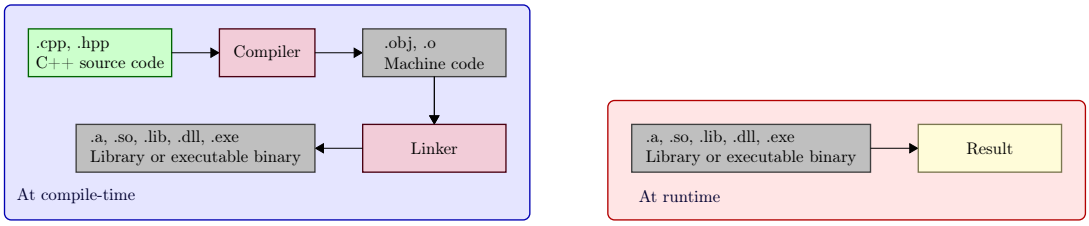
Many studies have been carried out to compare the advantages and disadvantages of each family of programming languages [4]. In this thesis we focus mainly on comparing the burden that are shouldered by the maintainer and the end-user.

5.1.1 Languages types

Compiled languages From the maintainer point of view, there is a lot of burden to shoulder. First he must decide whether he wants to distribute a package of source code or a package of binaries to the end-user. In the case of source code, he requires the end-user to have a compiler infrastructure, supported and validated by the library, on the end-user's machine, as well as all the source dependencies resolved via specific package managers such as cargo, conan or vcpkg, or via the system-wide package manager.

If the maintainer distribute binaries, he must generate one version for each couple of Operating System, Processor architecture he supports. Indeed, to generate a binary, there are many steps, as illustrated in fig. 5.1. First the compiler does a pass to generate machine code for each translation unit. There can be as many machine code as there are architecture and/or operating system supported. The maintainer may want to support different operating systems (last two Windows and OSX version, a handful Linux or Unix distributions, maybe mobile phone portages). Each of these OSes requires their own bundle. Additionally, the hardware may change, or the maintainer may want to take advantage of some specific hardware when available (like vectorized SIMD instructions such as SSE4, XOP, FMA4, AVX-512, etc.): this also requires the maintainer to multiply the number of binaries he compiles and distributes. Finally, the linker resolves the dependencies of the program and assembles the final binary. At that time the maintainer has to sort out how he wants to bundle the dependencies of his program. Should he statically link them alongside the binary and distribute them, at the risk of having the size of his binary exploding? Or should he state that the user has to install the dependencies on his system (via the system-wide package manager for instance) so that the binary can run? The burden of

handling the dependencies is then shifted to the end-user when installing the program. Usually the package manager, such as *apt*, *yum* or *pacman*, solves this transparently for the end-user and the programs works "out-of-the-box" once installed, as illustrated in fig. 5.1. The downside is that the maintainer has to publish many bundles of his package for each couple of Operating System, Processor architecture he supports.



(a) Compile-time (b) Runtime

Figure 5.1: Compiled languages: compile-time (a) vs. runtime (b).

Interpreted languages From the maintainer point of view, this is the ideal standpoint. It is easier to distribute software via an interpreted language because only the source code, a dependency tree and the assets are released in the distributed package. All the burden about resolving the dependencies and installing the framework to run the script is shouldered by the end-user that is using the program. Indeed, as shown in fig. 5.2, everything happens at runtime. The main advantage of this approach is to build a very rich ecosystem as distributing, maintaining and using programs is very easy once integrated in a package manager (often delivered alongside the language SDK natively, e.g. pip for Python). However, the most notable disadvantage is the performance which is explained by the fact that the source code is not compiled into optimized assembly code ready to be executed by the computer. Instead, the interpreter must do all the work in one go and, very often, this is slow (at least the first pass). Nowadays, interpreters differentiate two use cases. One is opening the console interpreter from the command line and typing commands to get the immediate interpreted results. This is called the read-eval-print-loop (REPL) [159]. This use-case usually does not provide heavy optimization because the user is likely prototyping his script and thus does not need it in the first place. The second use case is when the interpreter parses files and/or libraries as a whole. In this use-case, it is likely that the files do not change, while they are used a lot. It is then relevant for the interpreter to pay a pass to generate intermediate bytecode that can be interpreted faster for the future passes onward. As an example, the Python programming language has several implementations: CPython, PyPy, Jython or IronPython. CPython generate intermediate bytecode in `*.pyc` files while Jython, IronPython and PyPy embed a Just-In-Time (JIT) compiler to generate resp. JVM bytecode, CLR (.NET) bytecode, or a large variety of bytecode format.

Hybrid languages The burden is shared evenly between the maintainer and the user, while remaining minimal. Indeed, languages such as Java or C# are in this category. Those programming languages need a compiling pass which is designed to be fast, so that the feedback loop while prototyping remains fast. The result of the compilation is bytecode which is then executed on a hosting Virtual Machine (VM) that the user must install on his computer. The main advantages of this solution are the portability and the small distributed binary size. Indeed, in theory, any machine supporting the VM may also support the program. Also, as the VM executes the bytecode and resolves system dependencies, the distributed binary does not need to embed any system dependency. Finally, the user has the advantage of running a compiled program which provides fast user experience. The goal of hybrid languages is to bring together the advantages of both compiled and interpreted languages; no dependency management for the user, one small binary to distribute for the maintainer, good execution performance, and fast feedback loop

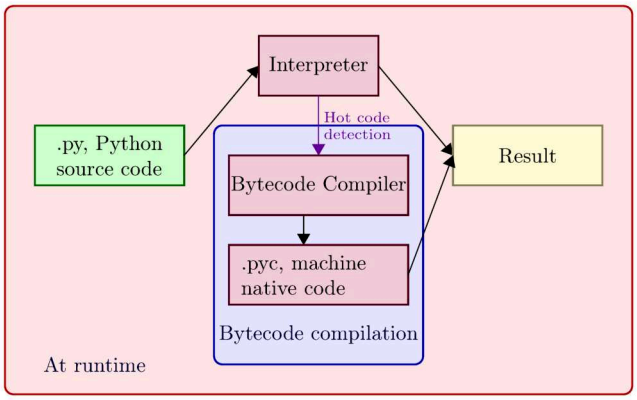


Figure 5.2: Interpreted languages: runtime

when prototyping (fast incremental compilation), while minimizing the downsides; usually a garbage collector is working inside the VM to handle memory allocations and de-allocations. In this regard, both Java and C# have achieved this feat quite elegantly. In theory, VM can further increase performance by implementing hot code detection which would compile the bytecode into native optimized machine code. This area is still a field of research to this day (cf. Java HotSpot [182, 79, 126]).

To summarize Interpreted and hybrid programming languages produces more portable artifacts and therefore are easier to deploy in a dynamic environment. We summarize in fig. 5.3 the different approaches in order to be able to execute a binary from source code.

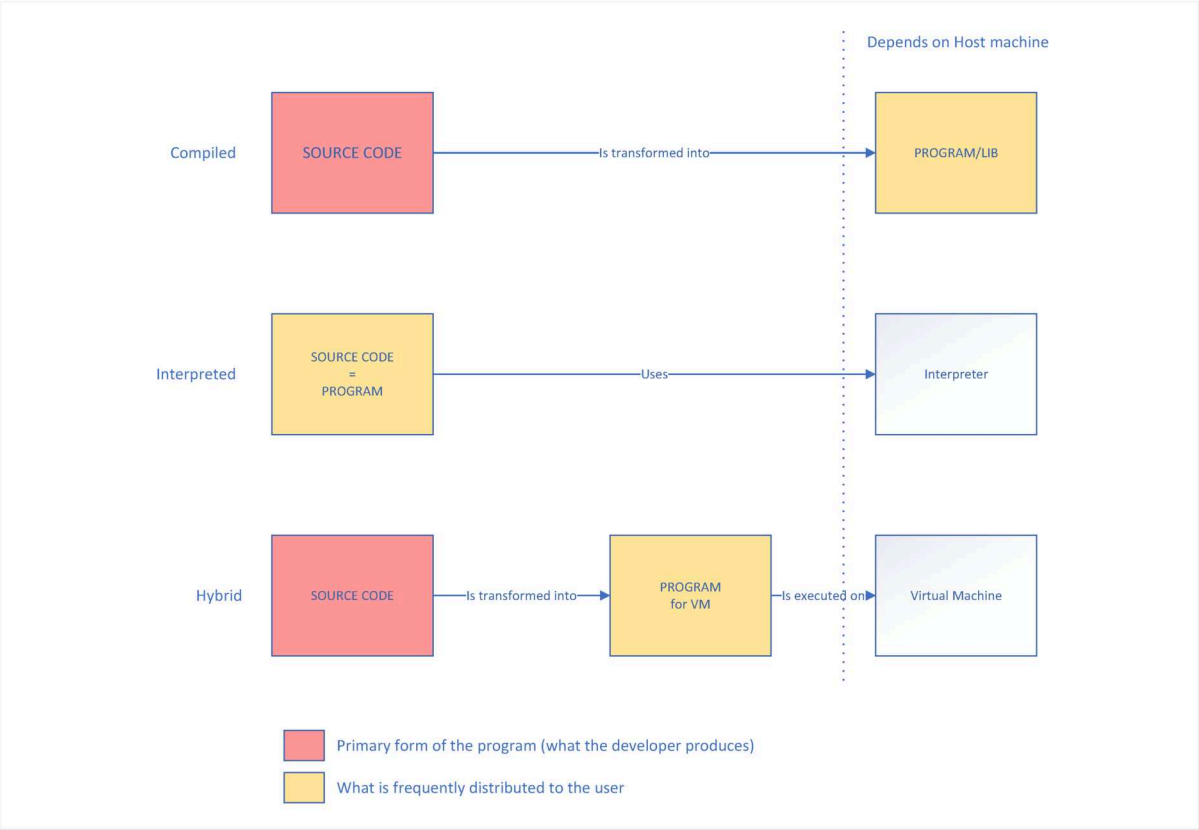


Figure 5.3: Languages types: summary diagram

5.1.2 Static and dynamic information

Compiled programming languages usually have poor support of introspection facilities. At best, static reflection is available at compile-time, but dynamic reflection is not an option. The structure of the program does not change at runtime. Some flexibility exists when delving into the area of hot-swapping dynamic libraries at runtime, however these techniques drag alongside security-related issues that injecting possible foreign machine code into one's program may generate. The only exception is Common Lisp. Indeed, there exists fully compliant implementation of this standard which are compiler-only, and runtime introspection is part of the Common Lisp standard. Interpreted programming languages usually have very developed introspection facilities. Dynamic reflection at runtime is possible and some language, such as Python (notably via the functions `dir` and `getattr`). Hybrid programming languages usually offers very good static and dynamic introspection facility at both compile-time and runtime, even if it means that runtime facilities will hurt performance. Also, those languages are usually designed to be able to hot-swap code at runtime. It is then possible to have the application running, recompile part of the binary of the application, replace the old running binary by the new compiled one, all at runtime.

The next important step is to classify what information is known at *compile-time* (machine/bytecode code generation): we call it *static* information; and what information is known at *runtime* (program execution): we call it *dynamic* information. In image processing, we have, on the one hand, knowledge about the following static information:

- 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.).

This means that, while the information may not be known at compile-time, we are able to write (or, more accurately, generate) code dedicated for those common types that we know constitute a large portion of the use cases. Furthermore, we can write optimized portion of code dedicated to handle some particular known types that the program will use when it encounters them at runtime. On the other hand, the following information are always dynamic:

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

The library needs both information type (static and dynamic) however, even if some information are missing, it is not a fatality and the library can still recover and work efficiently at runtime.

On another note, we notice that the architecture hosting the program is an information which is both static and dynamic. This translates the complexity of this information. Indeed, the maintainer needs to guess the array of architectures he wants to support and generate binaries for them (static). Also, the program needs to detect at runtime (dynamically) on which hardware he is running to possibly leverage it to increase performance. This is an area of research on its own called heterogeneous computing [146, 147].

5.1.3 Introducing our hybrid solution

Image processing communities like to have bridges with interpretable language such as Python or Matlab, to interface with their favorite tools, algorithms and/or facilities. As an example, with Python, the module NumPy [161] is a community standard which is heavily used. Henceforth, to broaden the usage of our library, we should be able to provide a way to communicate between our library and NumPy. There is always a need for genericity in both C++ and Python. Indeed,

```cpp
template <typename T>
T add(T a, T b) {
  return a + b;
}
```

(a) C++ static genericity

```python
def add(a, b):
  return a + b
```

(b) Python dynamic genericity

Figure 5.4: C++ Static (a) vs. Python Dynamic (b) genericity.

in C++ genericity is achieved via template programming, which is static, whereas in Python genericity is achieved via duck typing, which is dynamic, as shown in fig. 5.4.

On the one hand, static polymorphism induces no indirection in the generated code as the type is known at compile-time. It is then possible to generate optimized code for specific types. It is not possible to add a new supported type at runtime as the code has already been compiled. On the other hand, dynamic polymorphism implies that there will be some indirection when executing the code. Indeed, the code first needs to dispatch onto the appropriate function handling the input types to perform the operation properly. Nevertheless, it is possible to add new supported type at runtime without recompiling the library binary.

From the maintainer point of view, however, only distributing the C++ templated source code is a showstopper to the usability of his library by a Python user, because he does not hand over binaries. Indeed, one caveat of using C++ template programming is that the C++ compiler cannot generate a binary until it knows which type (of image, of value) will be used. But the maintainer does not know this information and the user (on Python's end) does not want to recompile the generic library code each time he has another set of types to try out. From here, there are still multiple ways to achieve our goal.

The first option is to embed and distribute, alongside the library, a JIT compiler whose job would be to generate the binaries and bindings just as they are used. This solution brings speed (excluding the first run that includes the compilation time) and unrestrained genericity. However, it binds both user and maintainer to the specificities of a compiler vendor, which means loosing in platform portability.

Another option is to type-erase (dynamic polymorphism) our types to enable the use of various concrete types through a single generic interface. This would translate into a class hierarchy whose concrete classes are the leaves (thus, whose value types and dimensions are known). This induces a non-negligible performance overhead but enables us to keep the genericity and portability at the cost of maintaining the class hierarchy.

Type generalization can also be considered. It is possible to cast everything into a super-type that is suitable for the vast majority of cases. For instance, we could say that we have a super-type `image4D<double>` into which we can easily promote subtypes such as `image2D<int>` or `image3D<float>`. Of course, we would lose the generic aspect and induce non-negligible speed cost whereas we would keep the platform portability.

And finally there is the dynamic dispatch. It consists in embedding dynamic information about types at runtime, and in dispatching (think of switch/case) to the correct facility that can handle those types. The obvious caveat is the cost of maintenance induced by the genericity as we would have numerous possible dispatches that would grow in a multiplicative way with the number of handled types, which is not very generic. On the other hand there is almost no speed loss and the portability is guaranteed. Theoretical models exist that could bring solutions to lower the number of dispatcher to write, such as multi-method [87]. Unfortunately they are currently not part of the C++ programming language.

5.2 Designing the hybrid solution

In Pylene we have chosen a hybrid solution midway between type-erasure and dynamic dispatch. The goal is to have a set of known types for which we have no speed cost, as well as supporting other types to ensure we remain generic. In [149] we provide a facility to expose our generic code to Python. As seen in the previous chapter, it is not possible to bind C++ source code to Python. We need to have a compiled binary implementing Python binding (we chose Pybind11 [137]) to be able to call C++ code from Python. In order to achieve the binding without sacrificing the genericity and the performance, we have designed a solution in two steps. We do not want to provide an abstract interface that will resolve the calls to access data on the call-site via virtual call because it would be very slow when the C++ code is executed. This would defeat the purpose of having to rely on C++ in a first place. However, it is possible to convert an abstract class into an instantiated concrete generic class whose template parameter are known. This requires, however, to enumerate all the possible cases. With modern C++, it has become possible to design $n \times n$ dispatch without gigantic switch-case clauses.

5.2.1 First step: converting back and forth

The first step of our solution consists in designing a buffer class that holds all the information about an image: dimension, underlying type, strides and pointer to data buffer. This class is named `ndimage_buffer`. When interfacing with Python, it is necessary to convert the Python image which is a `NumpPy.array` into our image type and vise versa, converting our C++ image type back into a Python image. The purpose of this buffer image is to hold all the information from the `NumpPy.array` to then instantiate a concrete C++ type. This process is illustrated in fig. 5.5. The first pitfall here is due to a limitation from the abstraction interface used in Python. Indeed, when using, for instance *Scikit-Image*, it is not possible to differentiate a 2D multichannel image from a 3D grayscale image because the image is always broken down to its most simple value and a 2D multichannel image is turned into a 3-dimensional `NumpPy.array` containing a single 8-bits channel, the last dimension contains only 3 elements at max but can theoretically contain more as the type system does not prevent that. To prevent this confusion, the C++ wrapper code may choose between two strategies; first is to consider all 3D/1-channel image as 2D/RGB images by default, second is to let the user give the information. For the sake of simplicity, we have chosen the first strategy.

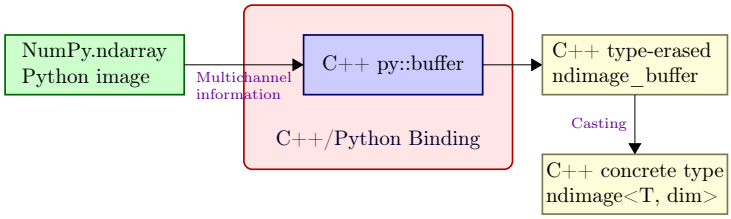


Figure 5.5: Bridge from Python to C++ via Pybind11 and a type-erased C++ class.

From the point of view of a practitioner, the code on the call-site (python side) should be as followed:

```
from skimage import data
import numpy as np
import Pylena as pln # our Python binding
img = data.astronaut() # 2D-rgb8 image -> NumPy.ndarray(ndim=3, dtype='uint8')
# pln.<any_algorithm>(img)
```

The C++ code contains lots of glue code necessary to expose the module to Python. In this thesis we have chosen to work with Pybin11 [137] which provides a modern API and is being actively maintained and improved. The glue code exposing the Python module from C++ is

given in appendix D.4. In order to have a seamless interaction between Python's `NumPy.ndarray` and C++, we need to define a proper strategy to convert the Python type into the C++ type without copying all data around. Pybind11 offers us two possibilities to achieve this. The first one is to use the buffer protocol to pass around NumPy's buffer information to C++ in a way so that C++ can properly interpret the data into a proper C++ class. The second one is to use a custom type-caster to implicitly convert the Python type into a C++ type each time it is needed.

With the first method, one would need to write the following code on Python's side:

```
np_img = data.astronaut() # 2D-rgb8 image -> NumPy.ndarray(ndim=3, dtype='uint8')
pln_img = pln.ndimage(np_img) # conversion into the C++ image type
pln_img_ret = pln.<any_algorithm>(pln_img) # call to any C++ algorithm
np_img_ret = pln_img.to_numpy(); # convert back into NumPy.ndarray
# use np.<...>(np_img_ret) # use resulting image with NumPy
```

Whereas the second method would require the user to only write the following code on Python's side:

```
np_img = data.astronaut() # 2D-rgb8 image -> NumPy.ndarray(ndim=3, dtype='uint8')
np_img_ret = pln.<any_algorithm>(np_img) # implicit conversion with custom type-caster on C++ side
# use np.<...>(np_img_ret) # use resulting image with NumPy
```

Removing this conversion step is the major reason we have chosen the second method: the custom type-caster. The C++ code for this part is given in appendix D.1.

We are now all set and are able to convert back-and-forth a Python image into a C++ image and vise versa.

5.2.2 Second step: multi-dispatcher (a.k.a. $n \times n$ dispatch)

The second step of our hybrid solution is to dispatch the abstract buffer type coming from Python to an efficient generic code. The naive way of doing so would be to include a gigantic switch-case clause in each algorithm implementation and dispatch to the correct instantiated generic algorithm from there. Aside from being a nightmare to maintain, the size of those clauses would grow several folds depending on the cardinality of the generic implementation. For instance, for a generic dilation, there are 3 axes of cardinality: the underlying type, the dimension and the structuring element shape. In the case where the library support 5 different structuring element shapes, 10 underlying types and 6 dimension for the image, the switch-case statement would need to dispatch over 300 clauses. Also, each supported algorithm would need to have dispatcher in their code. This solution defeats the purpose of genericity which is to write less code in the first place. We need to design a solution to implement those dispatchers while keeping our code short and efficient. The idea we took to solve this problem comes from the design of a C++ feature, the variant, and especially the visitor, applied to image processing, as in [89] for instance. We need to have a way to write the implementation of the algorithm once while enumerating all the possible cases. Also, if possible, the list of supported types should be written once at one place for maintenance purpose.

Simple dispatcher We then had the idea of writing a dispatcher. This dispatcher lists all the supported types and calls the given callbacks forwarding the given arguments by instantiating a specific type. Let us try to expose to Python, for instance, the generic existing algorithm for thresholding a binary image. The Python call-site code will look like this:

```
img_grayscale = skimage.data.grass()
pln.operators.binary_threshold(img_grayscale)
```

On the C++ side, we want to avoid writing code that looks like this:

```
mln::ndbuffer_image binary_threshold(mln::ndbuffer_image input) {
  auto dim = input.dim();
  auto tid = input.tid();
  switch(dim) {
```

```
      case 1: // 1D image
        switch(tid) {
          case UINT8:
            if(auto* image_ptr = input.template cast_to<uint8_t, 1>(); image_ptr)
              return mln::binary_threshold(*image_ptr);
          case RGB8:
            // error support only RGB8 images
        }
        break;
      case 2: // 2D image
        // ...
        break;
      // ... 3D, 4D, ...
    }
  }
```

Instead, it is possible to separate the dispatching code and the logical code entirely by using a templated operator, the same way we use lambdas in the pattern `std::variant`/`std::visit`. For our binary threshold example, the operator can be implemented just by writing the following code:

```
  // Operator templated by the dimension
  template <auto Dim>
  struct binary_threshold_op_t {
    // Function templated by the image type
    template <typename Img>
    mln::ndbuffer_image operator()(Img&& img) const {
      // Cast to a grayscale (information known) of the correct dimension
      if (auto* image_ptr = std::forward<Img>(img).template cast_to<std::uint8_t, Dim>(); image_ptr)
        // ACTUAL call to the generic algorithm
        return mln::binary_threshold(*image_ptr);
      else {
        std::runtime_error("Unable to convert the image to the required type.");
        return {};
      }
    }
  };
```

This code allows us to dispatch over any number of dimensions. We are required to pass a grayscale image for the algorithm so here the example is limited to dispatching over just one cardinality: the dimension. Let us now take a look at how we can implement the dispatcher for our example to work. The dispatcher must take the dimension as first parameter and any number of arguments to forward to the instantiated operator. The dispatcher then looks like the following code:

```
template <template <auto> class Op, typename... Args>
auto dispatch_v(std::size_t dim, Args&&... args) {
  switch (dim) {
    case (1):
      return Op<1>{}(std::forward<Args>(args)...);
    case (2):
      return Op<2>{}(std::forward<Args>(args)...);
    case (3):
      return Op<3>{}(std::forward<Args>(args)...);
    /* ... */
  }
}
```

The operator `Op` is instantiated with the correct dimension number. Then the `operator()` (parenthesis) is called while taking as parameters the forwarded arguments passed to the dispatcher. In our case, it will instantiate the type `binary_threshold_op_t<2>` and then call the function `binary_threshold_op_t<2>.operator()(input)`, forwarding the input image to the underlying algorithm. Indeed, using the dispatcher is as simple as writing `dispatch_v<binary_threshold_op_t>(input.dim(), input);`

The main advantage of this approach is that we respect all the requirements. First the logical code is bounded in the operator, second, the supported types are all listed in one place (the dispatcher), only once. Also, while our example is limited to one cardinality, any number of dispatcher can be piped to one after another to achieve the cardinality wanted.

Double dispatcher Let us push our example to implement the mathematical morphology dilation operator. We now have two more generic axes to cover: the structuring element shape and the underlying datatype. First, let us take a look at what the Python code may look like:

```
img_grayscale = skimage.data.grass()
rect = pln.se.rect2d(width=3, height=3)
img_dil = pln.operators.dilate(img_grayscale, se)
```

The first thing to notice is the need to add additional bindings to expose our C++ structuring elements. The glue code to achieve this is given in appendix D.2. Let us take a look at our dilation operator:

```
template <auto Dim, typename T>
struct dilate_operator_t {
  template <typename Img, typename SE>
  mln::ndbuffer_image operator()(Img&& img, SE se) const {
    if (auto* image_ptr = std::forward<Img>(img).template cast_to<T, Dim>(); image_ptr)
      // ACTUAL call to the generic algorithm
      return mln::dilation(*image_ptr, se);
    else {
      std::runtime_error("Unable to convert the image to the required type.");
      return {};
    }
  }
};
```

This operator needs double dispatch over two cardinalities: the dimension `Dim` and the value type `T`. We can skip the dispatch of the structuring element's shape as we have made a `std::variant` of all the supported structuring element for the sake of simplicity. Dispatching over the supported structuring elements can then be offloaded upstream from the call of the double dispatch, just by calling `std::visit`. We can immediately notice that there is an issue with our dilation operator. Indeed, there are two template parameters and our dispatcher `dispatch_v` does only handle one. We solve this issue by writing another intermediate operator dispatcher `dilate_operator_intermediate_t` serving as trampoline operator that will partially instantiate the final operator `dilate_operator_t` along the dimension template parameter to feed it to the last dispatcher, `dispatch_t`:

```
template <auto Dim>
struct dilate_operator_intermediate_t {
  template <typename Img, typename SE>
  mln::ndbuffer_image operator()(Img&& img, SE&& se) const {
    // Partial instantiation
    return double_dispatch_t<dilate_operator_t, Dim>(
            input.sample_type(), std::forward<Img>(input), std::forward<SE>(se));
  }
};
```

Dispatching the operator alongside two cardinalities (even three including the structuring element handled by `std::variant`) would then become as simple as calling:

```
// dispatch the structuring elements through using std::visit for std::variant
return std::visit(
[&input](const auto& se_) { // dispatch over the trampoline
    return dispatch_v<dilate_operator_intermediate_t>(input.dim(), input, se_);
  }, se);
```

In order for this to work, we need to piece together the final part of our puzzle, which is the double dispatch function that will handle the last dispatch along the underlying data while forwarding the first dispatch along the dimension. This dispatcher works the same as the simple one (`dispatch_v`) but take an additional template parameter (here `Dim`) that will be forwarded as-is to the given operator `Op`. The implementation then looks like this:

```
template <template <auto, typename> class Op, auto Dim, typename... Args>
auto double_dispatch_t(type_id tid, Args&&... args) {
  switch (tid) {
```

```
    case (type_id::INT8):
      return Op<Dim, std::int8_t>{}(std::forward<Args>(args)...);
    case (type_id::UINT8):
      return Op<Dim, std::uint8_t>{}(std::forward<Args>(args)...);
    case (type_id::DOUBLE):
      return Op<Dim, double>{}(std::forward<Args>(args)...);
      /* ... */
  }
}
```

We have now presented all the techniques and design required to build operators that are agnostic from the supported data-types, dimensions and/or additional data such as structuring elements. Indeed, the maintainer has gathered all the logic about listing the supported data types and dimension in one place: the custom dispatcher. He just needs to maintain those facilities to enable full support for all exposed algorithm, by default. This hybrid solution mixes type-erasure and modern C++ facilities to allow maximum performance. Indeed, the dispatch is performed before entering algorithms and the custom type-caster facility allows plugging C++'s image directly to the Python's image without having any unnecessary copies. The only caveat would be the code bloat incurred by all the explicit instantiation leading to compiling a larger and larger binary the more algorithms are being exposed. This can lead to performance issues due to potential pre-fetching memory optimization missed or code locality issues [42]. Another point not covered right now would be a way to support arbitrary data types, possibly injected from Python, into C++. Indeed, our hybrid solution only support the types provided by the library and listed in the dispatchers. It will instantiate all the code relative to them and support all the combinations, but the user may be tempted to plug a user-defined type from Python as an underlying image data-type. To allow this use-case, we introduce a new concept: the *value-set*. The value-set is a standard way to manipulate the underlying values. Through type-erasure, we can either manipulate known underlying value type with native facilities (near-zero overhead), or fallback to a virtual call that may report an error, or callback user-provided Python routine to manipulate unknown user values.

5.2.3 Third and final step: type-erasure & the value-set

As common thread in this section, we will work on the *stretch* algorithm which is naively defined in fig. 5.6. We can represent the pipeline calling this algorithm from Python in fig. 5.7. The motivation here is to abstract away the logic related to the underlying type inside the algorithm. In our example, it means reworking the lines 4 and 5 so that the value operations max and divide does not appear in our algorithm anymore. Instead, they will be delegated to a *value-set*. This would allow, theoretically, having all algorithm working for every underlying value type possible (be they static, dynamic, custom, injected from Python, etc.).

```
1 template <class T>
2 mln::image2d<float> naive_stretch(const mln::image2d<T>& src) {
3   mln::image2d<float> res  = mln::transform(src, [](auto val) -> float {
4     auto max = std::numeric_limits<T>::max();
5     return static_cast<float>(val) / static_cast<float>(max);
6   });
7   return res;
8 }
```

Figure 5.6: Stretch algorithm, naive C++ version.

Introducing the value-set The *value-set* is an abstraction layer around common operations needed when implementing an image processing algorithm such as an addition, a multiplication, a type conversion, getting the maximum, etc. It can be defined in C++ as a class template whose parameter is the manipulated type. The following code shows how to define a value-set:

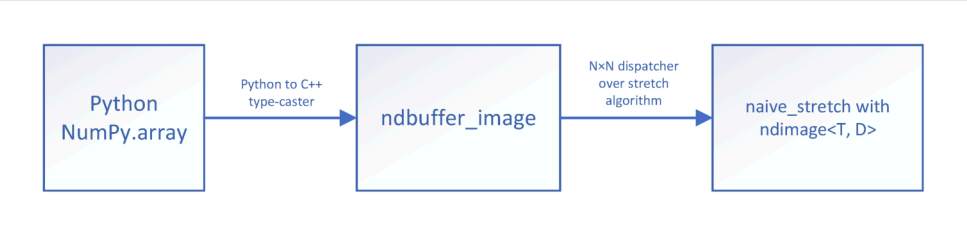


Figure 5.7: Python to C++ pipeline algorithm through the $n \times n$ dispatcher.

```
template <class T = void>
struct value_set {
  template <class U>
  U cast(T v) const { return static_cast<U>(v); }

  T max() const noexcept { return std::numeric_limits<T>::max(); }
  T min() const noexcept { return std::numeric_limits<T>::min(); }
  /* inf, sup, ... */

  T add(T l, T r) const noexcept { return l + r; }
  T sub(T l, T r) const noexcept { return l - r; }
  /* mod, pow, min, max, ... */
};
```

We can see that the default parameter of the class template is `void`. Indeed, we use the same technique as what is used in the standard library for all the comparison operators (`std:less`, `std::greater`, ...) which is known as the "diamond" [187, 188] operator (or transparent functions [189, 127]). It consists in providing a default (void) specialization that accept any types (by templates) and forward them directly to the underlying operator. The following code shows how to implement this specialization:

```
template <>
struct value_set<void> {
  template <class U, class T>
  U cast(T&& t) const { return static_cast<U>(std::forward<T>); }

  template <class T, class U>
  auto add(T&& l, U&& r) const noexcept { return std::forward<T>(l) + std::forward<U>(r); }
  template <class T, class U>
  auto sub(T&& l, U&& r) const noexcept { return std::forward<T>(l) - std::forward<U>(r); }
  /* ... */
};
```

The full code of the value-set is given in appendix D.3.2. The template parameter is shifted from the class to the member functions. It is also important to note that the member functions are not static, which requires to instantiate the `value-set` before using it. It may sound like a disadvantage at first glance, but it can be turned into an advantage later on. Indeed, this design allows a subclass to hold member variables which will be crucial for injecting user-types from Python.

Now that we have designed how our value-set is intended to work, we can deduce that an image is able to provide its own value-set. Indeed, an image knows what values it holds and thus is able to instantiate the proper value-set corresponding to this type. The member function returning the value-set in the class template `ndimage<T, D>` is then implemented as followed:

```
template <class T, std::size_t D>
class ndimage {
  /* ... */
  auto get_value_set() const noexcept {
    return value_set<T>{};
  }
};
```

Let us recall our naive stretch algorithm presented earlier. The pipeline representing the operations on values inside the algorithm is presented in fig. 5.8. Typically, this algorithm

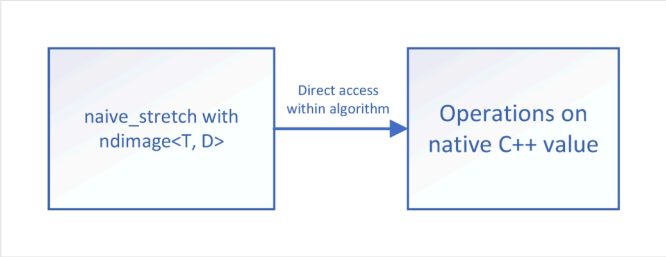


Figure 5.8: Naive stretch algorithm, pipeline to perform operations on values.

performs three operations that are the responsibility of a value-set: getting the max, performing a cast, and performing a division. The first step is then to use the value set shown above to abstract away those operations. The new algorithm is shown in fig. 5.9.

```
1  template <class T>
2  mln::image2d<float> fast_stretch(const mln::image2d<T>& src) {
3    auto                vs   = src.get_value_set();     // value-set for T
4    auto                vs_f = mln::value_set<float>{}; // fast value-set for float
5    mln::image2d<float> res  = mln::transform(src, [&vs, &vs_f](auto val) -> float {
6      auto max  = vs.max();                  // returns T
7      auto fval = vs.template cast<float>(val); // returns float
8      auto fmax = vs.template cast<float>(max); // returns float
9      return vs_f.div(fval, fmax);           // div directly returns float
10   });
11   return res;
12 }
```

Figure 5.9: Stretch algorithm, fast C++ version.

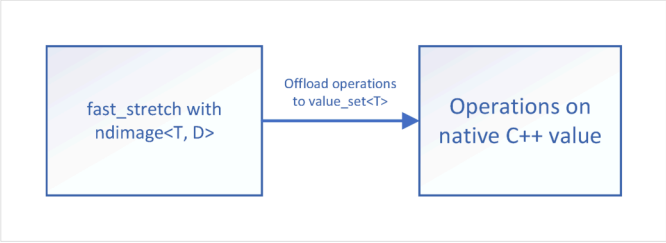


Figure 5.10: Fast stretch algorithm, pipeline to perform operations on values.

We instantiate the value-set needed for `T` values and for `float` values on lines 3 and 4. Then the maximum value is obtained via the value-set of `T` on line 6. Then a cast is performed via the value-set to get floating point value on lines 7 and 8. Finally, a floating point division is performed on line 9 via the formerly instantiates floating point value-set defined on line 4. Indeed, using the value-set instantiated for `T` would have performed a Euclidean division, which would have resulted in a bug. Calling this algorithm from Python is as simple as writing `ret = pln.fast_stretch(input)` once it is exposed. The pipeline representing the actual operations in the algorithm to access values is shown in fig. 5.10. The glue code to expose this algorithm is given in appendix D.3.3.

Type-erasure interface for value-set Moving one step further, we ultimately want to be able to dynamically inject a value-set into our algorithm. In order to achieve this feat, we need to design an abstract value-set as an interface so that a user can subclass it and provide his own value set. This abstract value-set needs to work with `std::any` as input value-type in order to provide a generic interface. Here, the genericity is achieved via type-erasure (the input value is type-erased into a `std::any`). The interface looks like the following code (full code is given in appendix D.3.2):

```
struct abstract_value_set {
  virtual ~abstract_value_set() {}

  virtual std::any max() const = 0;
  /* min, sup, inf, ... */

  virtual std::any add(const std::any& l, const std::any& r) const  = 0;
  virtual std::any div(const std::any& l, const std::any& r) const  = 0;
  /* sub, mult, ... */
};
```

The important part here is to notice that all values (returned and passed as argument) are now type-erased behind a `std::any`. From this interface, it is trivial to define the canonical subclasses for trivial types. We do this by defining the class template `concrete_value_set` which is able to generate a concrete interface for any given template type. The implementation will look like this:

```
template <typename T>
struct concrete_value_set : abstract_value_set {
  ~concrete_value_set() override {}

  template <class U>
  std::any cast(std::any v) const { return {static_cast<U>(std::any_cast<T>(v))}; }

  std::any max() const override { return {std::numeric_limits<T>::max()}; };
  /* min, sup, inf, ... */

  std::any add(const std::any& l, const std::any& r) const override {
    return {std::any_cast<T>(l) + std::any_cast<T>(r)};
  }
  std::any div(const std::any& l, const std::any& r) const override {
    return {std::any_cast<T>(l) / std::any_cast<T>(r)};
  }
  /* sub, mult, ... */
};
```

This concrete value-set implements simple dispatch via casting the type-erased `std::any` into the wanted value type to properly perform the operation. Thanks to this implementation, we are able to rewrite our stretch algorithm using this value-set to perform the value operations. The new algorithm is shown in fig. 5.11.

```
1   template <class T>
2   mln::image2d<float> virtual_dispatch_stretch(const mln::image2d<T>& src) {
3     auto                vs   = mln::concrete_value_set<T>{};     // value-set for T
4     auto                vs_f = mln::concrete_value_set<float>{}; // value-set for float
5     mln::image2d<float> res  = mln::transform(src, [&vs, &vs_f](auto val) -> float {
6       auto anymax  = vs.max();                                  // returns std::any
7       auto fanyval = vs.template cast<float>(val);              // cast to float in std::any
8       auto fanymax = vs.template cast<float>(anymax);           // cast to float in std::any
9       return std::any_cast<float>(vs_f.div(fanyval, fanymax)); // div returns float
10    });
11    return res;
12  }
```

Figure 5.11: Stretch algorithm, virtual dispatch version.

This algorithm uses two value sets (l.3 and l.4). One to get the maximum value corresponding to the original underlying value type in the image `T` (l.6), and to properly cast the values into float values (l.7–8). The other one is used to perform the floating point division (l.9). This algorithm is almost identical to the one given for `fast_stretch` earlier. Only the lines instantiating the value-set changes (l. 3–4) as well as the line handing over the result (l. 9) where we added a downcast from `std::any`. The pipeline presenting the operations needed to access values is presented in fig. 5.12.

Going forward: type-erasing the value entirely The need may arise where the user wants to handle images where the input value is abstracted away behind a dynamic type-erased type.

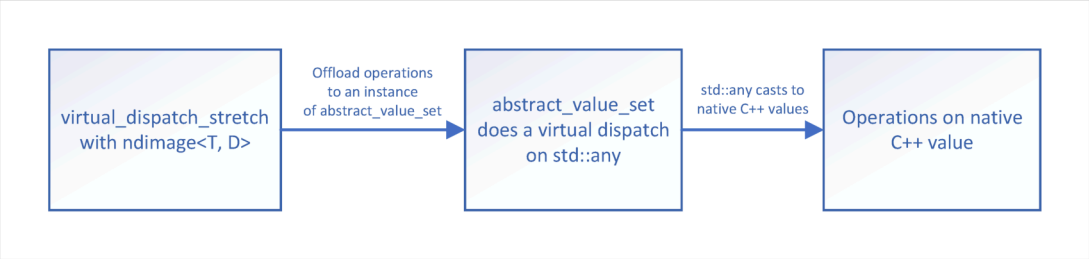


Figure 5.12: Virtual dispatch stretch algorithm, pipeline to perform operations on values.

This happens when handling image with heterogeneous value-type (different channel number from one value to another, optional information embed with the value, etc.). To handle this use-case, this value-type needs to be aware of its own value-set to perform the required operations. To achieve this, the value must embed the value set (as a pointer for instance) directly alongside the value. This embedding would allow writing such code possible:

```
class type_erased_value {
  type_erased_value add(const type_erased_value& rhs);
  /* ... */
};

void my_algo(ndimage<type_erased_value, Dim> img1,
             ndimage<type_erased_value, Dim> img2) {
  ndimage<type_erased_value, Dim> img_out = /* ... */;
  for (auto [v1, v2, out] : zip(img1.values(), img2.values(), img_out.values())) {
    out = v1.add(v2); // the value v1 knows how to perform the addition with v2
  }
}
```

This code allows supporting a fallback facility when all the supported values have been exhausted. For instance, when considering the previously defined `abstract_value_set`, and attempting to dispatch over all the supporting native C++ type unwrapped from the given `std::any`, if no type is matched, we can attempt one final unwrap to this type-erased value which is aware itself of how to perform the value operation. In pseudocode, it breaks down to the following logic:

```
1  std::any value_set<type_erased_value>::add(const std::any& lhs, const std::any& rhs){
2    abstract_value_set* vs = lhs.get_embedded_vs();
3    if (lhs.type() == typeid(int) && rhs.type() == typeid(int)) {
4      auto ret = std::any_cast<int>(lhs) + std::any_cast<int>; // unwrap, do the work
5      return std::any{ret}; // rewrap value
6    } else if (lhs.type() == typeid(float) && rhs.type() == typeid(float)) {
7      /* ... same ... */
8    } else {
9      auto te_lhs = std::any_cast<type_erased_value>(lhs); // last attempt
10     return lhs.add(rhs); // fallback on embedded value-set
11   }
12 }
```

First step is to conditionally attempt to cast the type-erased value over the supported native values (lines 3 and 6). When the type is supported then we unwrap it and perform the operation before rewrapping it in a `std::any`, and returning it. If we could not unwrap the value into a supported value-type, we make a last attempt on line 9 into our type-erased value-type. If this attempt succeeds, we rely on the fact that this value-type is aware of its own value-set and embed it to perform the required operation on line 10. The full code of the implementation for this value-set aware value-type is given in appendix D.3.2. We have relied on metaprogramming techniques in order to efficiently write the code that will do the work.

Thanks to this technique, we are able to write another version of our stretch algorithm which is shown in fig. 5.13.

This version is verbose because we do not yet support `mln::image2d<type_erased_value>` as a proper image type. Thus, we need to first convert the value of type `T` into the value-set

```cpp
template <class T>
mln::image2d<float> stretch_virtual_dispatch_type_erased_value(const mln::image2d<T>& src) {
  auto                vs    = mln::concrete_value_set<T>{};     // value-set for T
  auto                vs_f  = mln::concrete_value_set<float>{}; // value-set for float
  mln::image2d<float> res   = mln::transform(src, [&vs, &vs_f](auto val) -> float {
    // simulate having an image<type_erased_value>
    auto anyval = std::any{val}; // std::any of T
                                 // type_erased_value of std::any of T aware of value-set of T
    auto abs_anyval = mln::type_erased_value{anyval, vs};
    // instantiate a value-set for type_erased_value
    auto abs_vs = mln::value_set<mln::type_erased_value>{abs_anyval};
    auto anyabs_anymax =
        abs_vs.max(); // returns std::any of type_erased_value
                      // cast underlying std::any of type_erased_value of std::any of T into
                      // std::any of type_erased_value of std::any of float aware of value-set for float
    auto anyabs_fanyval = abs_vs.template cast<T, float>(std::any{abs_anyval}, &vs_f);
    auto anyabs_fanymax = abs_vs.template cast<T, float>(anyabs_anymax, &vs_f);
    // dispatch on known type, find a type_erased_value, then call
    // anyabs_fanyval.div(anyabs_fanymax) to perform division which will call
    // the underlying value-set for float for this operation
    auto anyabs_fanyret = abs_vs.div(anyabs_fanyval, anyabs_fanymax);
    // convert result back into float for returning to the image
    auto anyfret = std::any_cast<mln::type_erased_value>(anyabs_fanyret).val();
    return std::any_cast<float>(anyfret);
  });
  return res;
}
```

Figure 5.13: Stretch algorithm, virtual dispatch with a type-erased value version.

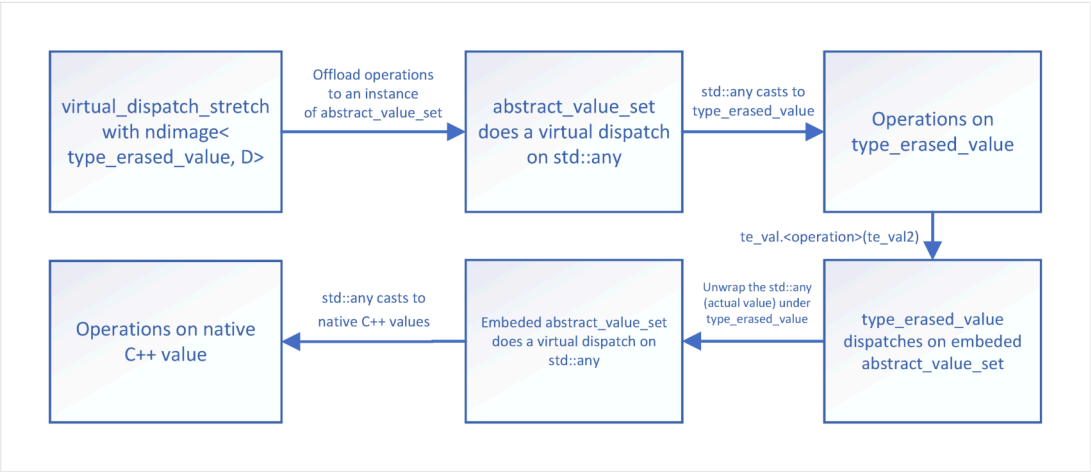


Figure 5.14: Virtual dispatch stretch algorithm with a type-erased value, pipeline to perform operations on values.

aware type-erased value type. This work is done from lines 7 to 11. Then on lines 12 to 21 we perform the actual algorithmic work on the values. This is where the dispatch mechanic, whose pipeline is shown in detail in fig. 5.14, will happen. Finally, on lines 23 and 24, we unwrap the previously wrapped type-erased value into a float to return for the algorithm.

Everything comes to a full circle: injecting a value-set from Python We have seen how to write an algorithm independently from its underlying types on C++ side thanks to relying on an abstraction layer: the value-set. This enables one fundamental feature which is code injection from Python. Indeed, every Python value is a `pybind11::object` (which is in itself a wrapper around the C-type `PyObject`), which is the generic way to refer to a non-trivially-convertible Python type we can write a value-set handling this value type. The value-set handling Python data-type provides the following interface:

```
1  template <>
2  struct value_set<pybind11::object> : abstract_value_set
3  {
4    value_set(pybind11::object python_vs_instance): vs_instance_(python_vs_instance) {}
5    ~value_set() override {}
6
7    template <typename U>
8    std::any cast(const std::any& v) const { /* ... */ }
9
10   std::any max() const override { {vs_instance_.attr("max")()}; }
11   /* min, sup, inf, ... */
12
13   std::any add(const std::any& l, const std::any& r) const override {
14     auto pyl = std::any_cast<pybind11::object>(l);
15     auto pyr = std::any_cast<pybind11::object>(r);
16     return {vs_instance_.attr("add")(pyl, pyr)};
17   }
18   std::any div(const std::any& l, const std::any& r) const override {
19     auto pyl = std::any_cast<pybind11::object>(l);
20     auto pyr = std::any_cast<pybind11::object>(r);
21     return {vs_instance_.attr("div")(pyl, pyr)};
22   }
23   /* max, min, sub, mult, ... */
24 private:
25   pybind11::object vs_instance_;
26 };
```

In this code we can clearly see at lines 4 and 25 that we are storing our Python's value-set instance into our class. This is possible due to the fact that our value-set abstraction is not providing static class function but member function. Hence, it is possible to offload the work of the value-set to a member variable at lines 10, 14–16 and 19–21 that will call the Python's value-set and get the wanted result. Also, at line 8 we use multiple techniques at once to get the correct resulting cast from a Python type. Indeed, the cast is done on Python side before being converted back into the corresponding C++ type. As such, we have to translate the wanted C++ type information into Python type information to request the cast. Once the cast is done, we need to unwrap the Python type into the corresponding C++ type and rewrap it into our, now favorite, `std::any`. The full implementation of this facility is given in appendix D.3.2.

On this particular matter, the user will find a Python abstract class to implement in order for his value-set to be usable by the library. This abstract class is defined by the following Python code:

```
# ABC stands for Abstract Base Class
class AbstractValueSet(ABC):  # AbstractValueSet is a metaclass that
                              # cannot be instantiated
  @abstractmethod # Forces the abstract semantic on the method: must be overridden
                  # in child classes
  def cast(self, value: Any, type_): pass
    if type_ in ["int", "float", "bool", "str"]:
      module = importlib.import_module('builtins')
```

```
      cls = getattr(module, type_)
      return cls(value)
    else:
      raise ValueError()
  @abstractmethod
  def max(self): return math.inf

  # ... min, sup, inf, ...

  @abstractmethod
  def add(self, lhs: Any, rhs: Any) -> Any: return lhs + rhs
  @abstractmethod
  def div(self, lhs: Any, rhs: Any) -> Any: return lhs / rhs

  # ... sub, mult, ...
```

This abstract class provides a facility to cast a value into a given type from its representation as a string. It also provides default/standard way of computing values. Those methods need to be overridden by a child class as they are all tagged with the `@abstractmethod` attribute. The full code of this data structure is given in appendix D.5.

Let us assume our user wants to build an image whose value type are his own custom Python data structure. For the sake of this example, we will name this specific value type `MyStruct`. The Python structure will look like this:

```
class MyStruct:
  v_: Any
  def __init__(self, v: Any): self.v_ = v
  def getV(self) -> Any:      return self.v_
  def setV(self, v: Any):     self.v_ = v
```

It is just a strong wrapper around a value with setters and getters. If we want to use this data structure as an image value-type and downstream in our C++ library, we need to provide a value-set which is aware of how to handle this value type properly. The user may want to write such a code:

```
img = np.array([MyStruct(1), MyStruct(2), MyStruct(6.5), MyStruct(3.14)] * 10, ndmin=2)
pln_img = pln.stretch(img) # Error, C++ does not know how to handle a value of type MyStruct
```

In this case, the user must provide his own value-set defined after the Python `AbstractValueSet` seen earlier. Such a value-set would look like this:

```
class MyValueSet(AbstractValueSet):
  def get_MyStruct__(self, v: Any):
    return v.getV() if isinstance(v, MyStruct) else v
  def cast(self, value: Any, type_):
    v = self.get_MyStruct__(value)
    return super().cast(v, type_)
  def max(self):
    return 255

  # ... min, sup, inf, ...

  def add(self, lhs: Any, rhs: Any) -> Any:
    l = self.get_MyStruct__(lhs)
    r = self.get_MyStruct__(rhs)
    return MyStruct(super().add(l, r))
  def div(self, lhs: Any, rhs: Any) -> Any:
    l = self.get_MyStruct__(lhs)
    r = self.get_MyStruct__(rhs)
    return MyStruct(super().div(l, r))

  # ... sub, mult, ...
```

The full code of this implementation is given in appendix D.5. With this information, we can now write the following Python code:

```
img = np.array([MyStruct(1), MyStruct(2), MyStruct(6.5), MyStruct(3.14)] * 10, ndmin=2)
pln_img = pln.stretch(img, value_set=MyValueSet()) # This works !
```

And this works fine.

On the C++ side, the maintainer just needs to write another version of the stretch algorithm supporting the Python value-set which is shown in fig. 5.15.

```
1   template <class T>
2   mln::image2d<float> slow_stretch(const mln::image2d<T>& src, const mln::value_set<pybind11::object>& py_vs) {
3     mln::image2d<float> res = mln::transform(src, [&py_vs](auto val) -> float {
4       auto anymax  = py_vs.max();                        // returns std::any of pybind11:object
5       auto anyval  = std::any{pybind11::cast(val)};      // converts to std::any of pybind11::object
6       auto anyret  = py_vs.div(anyval, anymax);          // returns std::any of pybind11::object
7       auto anyfret = py_vs.template cast<float>(anyret); // returns std::any of float
8       return std::any_cast<float>(anyfret);              // returns float
9     });
10    return res;
11  }
```

Figure 5.15: Stretch algorithm, injected value-set from Python version.

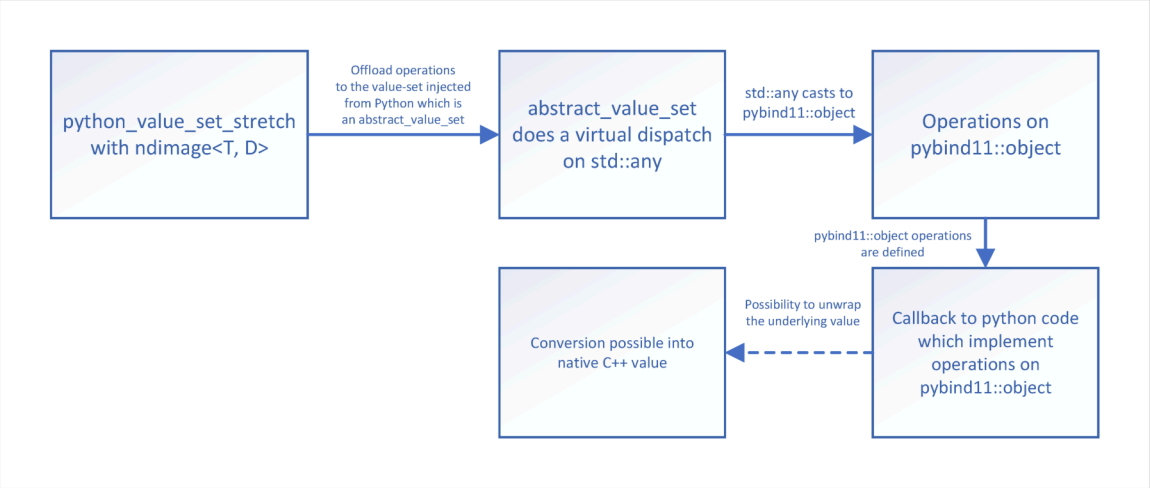


Figure 5.16: Stretch algorithm with an injected value-set from Python code, pipeline to perform operations on values.

The `mln::value_set<pybind11::object>` is constructed from the passed `value_set` argument on python side as it is a `pybind11::object`. The dispatcher then forward it to the algorithm as an additional parameter. We then have actual Python call on lines 4, 6 and 7. The line 5 just wrap the image value into a Python object so that it can then be forwarded to the Python code upstream, as described in the pipeline shown in fig. 5.16. The line 7 casts the Python type into an actual C++ type (wrapped into our `std::any` wrapper). Finally, the line 8 unwrap the float value to return it to the algorithm.

5.3 Summary and continuation

5.3.1 Performance & overhead

In this chapter, we have designed many solutions to solve several kinds of issues related to injecting dynamic code in our generic code. However, layering one abstraction layer after another, or even calling Python code does come with performance cost. This is why we have run a benchmark to outline the cost of the proposed solutions. The full code of the benchmark is given in appendix D.6. This benchmark compares the four version of our stretch algorithm. The result is shown in table 5.1.

Dispatch type	Compute Time	ΔCompute Time
Native value-set with native C++ value-type (baseline) 5.9	0.0093s	0
Value-set with virtual dispatch with native C++ value-type 5.11	0.1213s	×13
Value-set with virtual dispatch with C++ type-erased values 5.13	1.0738s	×115
Injected Python value-set with native C++ value-type 5.15	21.5444s	×2316

Table 5.1: Benchmarks of all our version of the stretch algorithm.

This benchmark shows that each time an abstraction layer is added on top of the baseline, the user must expect a 10× slowness factor for his code performance. Also, calling Python code is immensely slower (2300× !) than the baseline. This renews the interest there is to recompile the templated C++ library with an additional known type rather than injecting it from Python code. Being able to inject dynamic Python code eases prototyping and increase the speed at which the user can write his code. However, the benchmark shows that this is not a viable solution once the prototype needs to upgrade into to a production-ready program.

5.3.2 Continuation: JIT-based solutions, pros. and cons.

Our hybrid solution certainly has advantages, however the main disadvantage is the slowness of injecting our own types from the Python side. There exists another solution that this thesis did not have the opportunity to study in-depth. This solution is based on a well-known technology: the Just-In-Time (JIT) compilation which has been previously illustrated in fig. 5.2 (and which itself is based on the notion of generative programming [30]). Library such as AsmJit [91] are able to emit machine code directly by making call in C++ code. JIT is a technology already used by interpreted languages such as Java or PHP to generate on-the-fly native and optimized machine code for the section of the source code that is considered "hot" by the interpreter. A source code is "hot" when it is executed a lot: the end-user would gain paying the compilation time once to have this code executed faster several times later on. When applying this strategy to our problematic, it would mean that the user must be able to compile native machine code from the templated generic C++ code by injecting the requested type when it is used. Such an operation shifts a heavy burden on the user, and it is well-known that compiling C++ code is notably *complicated* and *slow*. In addition, the library needs to be able to auto-generate Python binding once the C++ code is compiled, and to handle *NumPy.ndarray* types in the interface. There are several solutions to achieve this process.

The first solution is to use basic system call to the compilers to actually *compile* C++ code once the templated types are known and explicitly instantiated in the source code. This solution requires careful code-generation design and that the user actually possess a compiling infrastructure on his computer. Furthermore, the user must resolve all the library dependencies, such as *freeimage* for IO etc. This solution is engineered in the VCSN library [108]. Indeed, each time the user declares a new automaton in his Jupyter notebook, corresponding source code is compiled in the background and then cached. It is a very perilous solution to implement when the final execution environment (OS, installed software) is not well-known in advance. Nowadays, the issue may be lesser thanks to containerization, however, it still requires to maintain both the library and the container solution to use it (Docker, Rocket, etc.).

The second solution is to use Cython [88]. It is a transpiling infrastructure which transforms a Python source code directly into C-language source code so that it can be compiled by a standard C compiler just by linking against the Python/C API. This remove the burden of writing the careful code-generation routine, system-calls to the C++ compiler and removes the need to resolve all the dependencies. This infrastructure takes care of everything for the user. Cython even support C++ template code [176] which is mandatory for our use-case.

The third solution consists in relying on recent projects that are all relying on the LLVM infrastructure. We can notably note AutoWIG [141], Cppyy [130], Xeus-cling [173] and Pythran [123].

AutoWIG has in-house code based on LLVM/Clang to parse C++ code in order to generate and compile a Swig Python binding using the Mako templating engine. AutoWIG, coupled with Cython would permit the user to, for instance, generate C code related to a custom Python structure. Then a simple call to AutoWIG will parse the C code and inject it into the C++ library to generate the appropriate bindings for the user. As for Cppyy, it is based on LLVM/Cling, a C++ interpreter, and can directly interpret C++ code from a Python string. This enables injecting custom types easily, be they in Python code (transpiled by Cython) or C++ code (directly interpreted by Cling). Afterwards, the infrastructure generates the appropriate binding from the templated C++ library for the injected type. Xeus-cling is a ready-to-use Jupyter kernel and allow the usage of C++ code directly from within a notebook. This completely bypass the need of Python bindings in the first place and allow the user to use the library from within the notebook as if he was using a Python library. Finally, Pythran is an ahead of time compiler for a subset of the Python language, with a focus on scientific computing. It takes a Python module annotated with a few interface descriptions and turns it into a native Python module with the same interface, but hopefully faster. Pythran takes advantages of multicore and SIMD instructions to turns its subset of the Python language into heavily templated C++ code instantiated for particular optimized types. All those infrastructures, however, come with a hefty cost in terms of binary size. Indeed, a C++ compiler is not small and embarking it alongside the image processing library can easily impact greatly the final binary. For instance, without the LLVM infrastructure the binary would weight around 3MB. However, with the LLVM infrastructure, the binary would weight at the bare minimum 50MB. Also, these solutions may not be immediately faster. Indeed, when prototyping back and forth with a variety of types, the user may not be eager to wait for long compilations times each time he is testing a new iteration of his work. Despite those facts, those solutions offers great avenue of research for the future and the author is eager to explore them.

Chapter 6

Conclusion and continuation

The contributions presented in this thesis by the author followed a very clear narrative arc. The emphasis was put first on presenting the notion of generic programming (genericity), its story and how anyone can relate in his day-to-day life, in particular when applied to image processing. Genericity is a 4-decades years old notion that has evolved and been adopted in very modern areas of our society. Indeed, image processing is widely used to build modern applications used all around the world. However, it was demonstrated how difficult it can be to implement solutions relying on genericity. Indeed, there is a rule of three tying genericity, performance and ease of use. The rule states that one can only have two of those items by sacrificing the third one. If one wants to be generic and efficient, then the naive solution will be very complex to use with lots of parameters. If one wants a solution to be generic and easy to use, then it will be not very efficient by default. Finally, if one wants a solution to be easy to use and efficient then it will not be very generic. In this thesis, we endeavor to demonstrate how to break through this rule by following three steps.

The first step, illustrated in chapter 3, was to create an inventory of image types families as well as image processing algorithms families. The aim was to produce a comprehensive taxonomy of types (pixel, image, structuring elements, ...) and algorithms related to image processing in order to be able to write concepts (in the sense of C++ concepts). This first step delimits the perimeter of what the author means by *genericity*. From this starting point, it becomes easier to write image processing algorithms, just by relying on those concepts. Furthermore, different concepts exist to enable algorithm implementers to leverage properties (structuring elements' decomposability, image's buffer contiguity, ...) in order to achieve maximum performance.

At this point, we need to raise the reasoning level from being at the layer of pixels (low) to being at the layer of images (higher), in order to allow fast prototyping for simple operations while guarantying very small memory footprint and near-zero performance impact. For this reason, we expand the concept of *views* from the C++ standard (2020) to images, and we design the notion of *image view*. We also make the design choice to have cheap-to-copy image (with a shared data buffer and reference semantic) by default in order to merge concrete image and views from the user point of view. The lazy-evaluation, that systematically happens when using views allows performance gain, e.g. when clipping larges images. In the case where the whole image is processed, we were able to still retain very satisfactory performances that remain stable. Also, we show through concrete use-case, such as pixel-wise algorithm and border management how the usage of views simplify greatly how to write more complex image processing algorithms, that are efficient by default. We finally show the limitations of this approach, with a particular focus on the speed of traversing an image which is a mandatory use-case we must get right.

Finally, this thesis focused on how it is possible to distribute this software to the image processing community, which is mainly working with Python. The last work concentrates its efforts on finding the best way to design a static (compile-time, templated C++) — dynamic (runtime, Python notebook) bridge to bring those notions (concepts and views) to the practitioner,

efficiently. This last work explores this dilemma and offers to address it with a hybrid solution whose design is explained in-depth. This hybrid solution relies on a type-erased type which offers compatibility with a *NumPy.ndarray*. It is then able to cast itself inside an $n \times n$ dispatcher (dimension and underlying type) into an optimized concrete templated C++ type. This solution also explains how to write very simply the glue code enabling already-existing algorithms (in C++) to be exposed in Python thanks to a dispatch mechanic heavily inspired from the C++ standard (`std::visit`, `std::variant`). The aim of this solution was to regroup at a single place in the code all the supported types into the dispatchers for maintenance purpose, as well as demanding minimal work from algorithm implementer to expose their algorithms, all this while keeping the native performance. Indeed, no superfluous copy is performed thanks to *pybind11*'s *type-caster* facility, and one cast is done from the type-erased type to the native one. All the instructions inside the algorithm are performed on native, optimized type. Finally, this solution offers a way to inject custom Python types into the library for prototyping purpose, thanks to a new abstraction layer, but at the cost of heavy performance loss. The downside of this bridge is obviously the code bloat in the resulting image processing library binary whose size explodes exponentially with the number of supported types multiplied by the number of algorithms multiplied by the number of additional supported data (structuring elements, label map, etc.)

We conclude this thesis by offering a new avenue of research around the Just-In-Time (JIT) compilation area to further improve the bridge between the static and the dynamic world. The author thinks that this avenue is worth exploring, especially with the already promising existing tools (Xeus-cling [173], Cppyy [130], Cython [88], AutoWIG [141], Pythran [123]) in order to solve the code bloat issue. Indeed, we would only compile what the user needs, but the entry price may be to statically link a C++ interpreter (LLVM/Cling?) into the image processing library binary which in itself would greatly bloat it. It may be possible, however, to rely on the user's system-wide infrastructure so that the packaged image processing library does not distribute a whole C++ interpreter/compiler alongside the image processing library binary. This is still a new area of research and the author would very much want to delve into it to study further what is possible to achieve as of today with those tools for the image processing community.

In addition, it would be interesting to expand the Concept framework presented in chapter 3 to encompass other areas related to computer vision, such as 3D computer graphics (CGI), visualization of 3D scenes in real time or augmented reality. Those areas are booming and are in dire need of performance. We feel that such a framework would be useful to improve usability of existing libraries while improving performances.

Also, it would be interesting to question if there are other programming languages, other than C++, with the required generic capabilities for image processing and performances. For example, Rust's generic facilities, which are trait-based with a `where` clause, seems to be promising. Indeed, C++ has a heavy governance with inertia (it is an ISO standard) and the language still misses some major key features such as reflection, introspection and contracts. It took 20 years to get Concepts into the standard. The state of tooling and package management seems to be, to this day, very clunky. There are lots of tools to do the same things in different ways, with different level of compatibility and platform support. By contrast, Rust has a full integrated environment, and it is very pleasant to start a project with it.

To finish with programming-related research avenue, it would be interesting to do a particular focus on heterogeneous computing applied to algorithms canvas. Indeed, we have presented how, in theory, we can factorize optimization opportunity with canvas. We now need to leverage this opportunity to make it possible to offload work on, for instance, GPU or cloud infrastructure.

Another research avenue would be to find a way to validate our assertion about the library's code being easy to use in an empirical way. Indeed, we rely on notion such as code clarity or code expressiveness, but those notions are subjective. As we are expert in our area, our point of view about what is friendly and/or expressive is biased. It would be interesting to make an

experiment with students learning how to program, where one student pool is given a library with inexpressive / unfriendly code (in our point of view), and another student pool is given a library with expressive / friendly code (in our point of view) to measure how fast they are able to program complex applications with it. If we repeat this experiment over time, following a methodology derived from clinical trial (for ethic and preventing bias), we will be able to empirically validate our assertion about the friendliness and expressiveness of code.

Part IV

Appendices

Appendix A

References

Bibliography

[1] Russell R. Atkinson, Barbara H. Liskov, and Robert W. Scheifler. "Aspects Of Implementing CLU". In: *Proceedings of the 1978 Annual Conference*. ACM '78. Washington, D.C., USA: Association for Computing Machinery, 1978, pp. 123–129. ISBN: 0897910001. DOI: 10.1145/800127.804079. URL: https://doi.org/10.1145/800127.804079 (cit. on pp. 66, 79).

[2] John Backus. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: https://doi.org/10.1145/359576.359579 (cit. on pp. 55, 56).

[3] ANSI. *ANSI/MIL-STD-1815A: Programming languages — Ada*. American National Standards Institute, June 1983, p. 333. URL: https://www.iso.org/standard/16028.html (cit. on p. 66).

[4] Barry W. Boehm. "Software Engineering Economics". In: *IEEE Transactions on Software Engineering* SE-10.1 (Jan. 1984), pp. 4–21. ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010193 (cit. on p. 129).

[5] Joseph A. Goguen. "Parameterized Programming". In: *IEEE Transactions on Software Engineering* SE-10.5 (Sept. 1984), pp. 528–543. ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010277 (cit. on p. 56).

[6] ISO. *ISO/IEC JTC 1/SC 22: Programming languages — Ada*. Geneva, Switzerland: International Organization for Standardization, June 1987, p. 333. URL: https://www.iso.org/standard/16028.html (cit. on p. 66).

[7] Alexander A Stepanov, Aaron Kershenbaum, and David R Musser. *Higher order programming*. March, 1987 (cit. on p. 56).

[8] David R. Musser and Alexander A. Stepanov. "Generic programming". In: *Intl. Symp. on Symbolic and Algebraic Computation*. Springer. 1988, pp. 13–25 (cit. on pp. 11, 26, 56, 59, 66, 72, 73).

[9] David R Musser and Alexander A Stepanov. *The Ada Generic Library linear list processing packages*. Springer-Verlag, 1989 (cit. on p. 56).

[10] Gerhard X. Ritter, Joseph N. Wilson, and Jennifer L. Davidson. "Image algebra: An overview". In: *Computer Vision, Graphics, and Image Processing* 49.3 (1990), pp. 297–331 (cit. on p. 74).

[11] Marcel van Herk. "A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels". In: *Pattern Recognition Letters* 13.7 (1992), pp. 517–521. ISSN: 0167-8655. DOI: https://doi.org/10.1016/0167-8655(92)90069-C. URL: http://www.sciencedirect.com/science/article/pii/016786559290069C (cit. on p. 88).

[12] R. Adams. "Radial Decomposition of Disks and Spheres". In: *CVGIP: Graphical Models and Image Processing* 55.5 (1993), pp. 325–332. ISSN: 1049-9652. DOI: https://doi.org/10.1006/cgip.1993.1024. URL: https://www.sciencedirect.com/science/article/pii/S1049965283710242 (cit. on p. 88).

[13] Florian Cajori. *A history of mathematical notations*. Vol. 1. Courier Corporation, 1993 (cit. on p. 87).

[14] Barbara Liskov. "A History of CLU". In: *SIGPLAN Not.* 28.3 (Mar. 1993), pp. 133–147. ISSN: 0362-1340. DOI: 10.1145/155360.155367. URL: https://doi.org/10.1145/155360.155367 (cit. on p. 65).

[15] David R. Musser and Alexander A. Stepanov. "Algorithm-oriented generic libraries". In: *Software: Practice and Experience* 24.7 (1994), pp. 623–642. DOI: 10.1002/spe.4380240703. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380240703. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380240703 (cit. on pp. 10, 24, 57, 59, 72).

[16] Bjarne Stroustrup. *The Design and Evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994. ISBN: 0-201-54330-3 (cit. on p. 72).

[17] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. "Curry: A Truly Functional Logic Language". In: *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*. Vol. 95. 1995, pp. 95–107 (cit. on pp. 15, 30, 124).

[18] ISO. *ISO/IEC JTC 1/SC 22: Programming languages — Ada*. Geneva, Switzerland: International Organization for Standardization, Feb. 1995, p. 511. URL: https://www.iso.org/standard/22983.html (cit. on p. 66).

[19] Alexander Stepanov and Meng Lee. *The standard template library*. Vol. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995 (cit. on p. 56).

[20] Todd Veldhuizen. "Expression templates". In: *C++ Report* 7.5 (1995), pp. 26–31 (cit. on pp. 16, 31, 51, 55, 124).

[21] David M Beazley et al. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++." In: *Tcl/Tk Workshop*. Vol. 43. 1996, p. 74 (cit. on p. 78).

[22] James O. Coplien. "Curiously Recurring Template Patterns". In: *C++ Gems*. USA: SIGS Publications, Inc., 1996, pp. 135–144. ISBN: 1884842372 (cit. on p. 70).

[23] Ronald Jones and Pierre Soille. "Periodic lines: Definition, cascades, and application to granulometries". In: *Pattern Recognition Letters* 17.10 (1996), pp. 1057–1063. ISSN: 0167-8655. DOI: https://doi.org/10.1016/0167-8655(96)00066-9. URL: https://www.sciencedirect.com/science/article/pii/0167865596000669 (cit. on p. 88).

[24] G. S. Novak. "Software reuse by specialization of generic procedures through views". In: *IEEE Transactions on Software Engineering* 23.7 (July 1997), pp. 401–417. ISSN: 2326-3881. DOI: 10.1109/32.605759 (cit. on pp. 13, 28, 105).

[25] ISO. *ISO/IEC 14882:2003: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Sept. 1998, p. 732. URL: https://www.iso.org/standard/25845.html (cit. on pp. 56, 66, 67, 70).

[26] Todd L. Veldhuizen. "Arrays in Blitz++". In: *Computing in Object-Oriented Parallel Environments*. Ed. by Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 223–230. ISBN: 978-3-540-49372-3 (cit. on pp. 51, 55).

[27] Bjarne Stroustrup. "An overview of the C++ programming language". In: *Handbook of Object Technology* (1999) (cit. on p. 72).

[28] Matthew H. Austern. "Segmented Iterators and Hierarchical Algorithms". In: *Generic Programming*. Ed. by Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 80–90. ISBN: 978-3-540-39953-7 (cit. on p. 120).

[29] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000) (cit. on pp. 8, 22, 51, 52, 61).

[30] Krzysztof Czarnecki et al. “Generative Programming and Active Libraries”. In: *Generic Programming*. Ed. by Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 25–39. ISBN: 978-3-540-39953-7. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.646.8822&rep=rep1&type=pdf (cit. on pp. 17, 33, 147).

[31] James C. Dehnert and Alexander Stepanov. “Fundamentals of Generic Programming”. In: *Generic Programming*. Vol. 1766. LNCS. Springer. 2000, pp. 1–11 (cit. on pp. 10, 24, 56, 57, 59, 72, 73).

[32] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography”. In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035. URL: https://doi.org/10.1145/352029.352035 (cit. on pp. 8, 22, 51).

[33] Alexandre Duret-Lutz. “Olena: a component-based platform for image processing, mixing generic, generative and OO programming”. In: *symposium on Generative and Component-Based Software Engineering, Young Researchers Workshop*. Vol. 10. Citeseer. 2000 (cit. on p. 72).

[34] Thierry Géraud and Alexandre Duret-Lutz. “Generic programming redesign of patterns”. In: *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee, Germany: UVK, Univ. Verlag, Konstanz, July 2000, pp. 283–294 (cit. on p. 61).

[35] Thierry Géraud et al. “Obtaining genericity for image processing and pattern recognition algorithms”. In: *Proceedings of the 15th International Conference on Pattern Recognition (ICPR)*. Vol. 4. Barcelona, Spain: IEEE Computer Society, Sept. 2000, pp. 816–819 (cit. on pp. 72, 74).

[36] Øyvind Kolås and et al. *Generic Graphic Library*. Available at http://www.gegl.org. 2000 (cit. on p. 51).

[37] Ullrich Köthe. “STL-Style Generic Programming with Images”. In: *C++ Report Magazine* 12.1 (2000). Available at https://ukoethe.github.io/vigra, pp. 24–30 (cit. on pp. 51, 61, 78).

[38] L. Prechelt. “An empirical comparison of seven programming languages”. In: *Computer* 33.10 (Oct. 2000), pp. 23–29. ISSN: 1558-0814. DOI: 10.1109/2.876288 (cit. on pp. 16, 32, 129).

[39] Jeremy Siek and Andrew Lumsdaine. “Concept checking: Binding parametric polymorphism in C++”. In: *First Workshop on C++ Template Programming*. Citeseer. 2000 (cit. on p. 71).

[40] Todd L. Veldhuizen. “Blitz++: The Library that Thinks it is a Compiler”. In: *Advances in Software Tools for Scientific Computing*. Ed. by Hans Petter Langtangen, Are Magnus Bruaset, and Ewald Quak. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 57–87. ISBN: 978-3-642-57172-5 (cit. on pp. 16, 31, 51, 55, 124).

[41] M. Allili, K. Mischaikow, and A. Tannenbaum. “Cubical homology and the topological classification of 2D and 3D imagery”. In: *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*. Vol. 2. Oct. 2001, 173–176 vol.2. DOI: 10.1109/ICIP.2001.958452 (cit. on p. 88).

[42] Abdel-Hameed A. Badawy et al. “Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations”. In: *Proceedings of the 15th International Conference on Supercomputing*. ICS ’01. Sorrento, Italy: Association for Computing Machinery, 2001, pp. 486–500. ISBN: 158113410X. DOI: 10.1145/377792.377906. URL: https://doi.org/10.1145/377792.377906 (cit. on p. 138).

[43] ISO. *ISO/IEC JTC 1/SC 22: Programming languages — Ada — Technical Corrigendum 1*. Geneva, Switzerland: International Organization for Standardization, June 2001, p. 56. URL: https://www.iso.org/standard/35451.html (cit. on p. 66).

[44] Andrew Lumsdaine Jeremy Siek Lie-Quan Lee. "The Boost Graph library". In: Addison-Wesley, 2001. URL: http://cds.cern.ch/record/1518180/files/0201729148_TOC.pdf (cit. on p. 94).

[45] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001. URL: http://www.scipy.org/ (cit. on p. 51).

[46] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151 (cit. on p. 55).

[47] Jérôme Darbon, Thierry Géraud, and Alexandre Duret-Lutz. "Generic implementation of morphological image operators". In: *Mathematical Morphology, Proceedings of the 6th International Symposium (ISMM)*. Sydney, Australia: CSIRO Publishing, Apr. 2002, pp. 175–184 (cit. on pp. 72, 74).

[48] Delores M Etter, David C Kuncicky, and Douglas W Hull. *Introduction to MATLAB*. Prentice Hall, 2002 (cit. on p. 62).

[49] David Vandevoorde and Nicolai M Josuttis. *C++ Templates: The Complete Guide, Portable Documents*. Addison-Wesley Professional, 2002 (cit. on p. 69).

[50] Tomihisa Welsh, Michael Ashikhmin, and Klaus Mueller. "Transferring Color to Grayscale Images". In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, Texas: Association for Computing Machinery, 2002, pp. 277–280. ISBN: 1581135211. DOI: 10.1145/566570.566576. URL: https://doi.org/10.1145/566570.566576 (cit. on p. 112).

[51] Terry S Yoo et al. "Engineering and algorithm design for an image processing API: a technical report on ITK-the insight toolkit". In: *Studies in health technology and informatics* (2002), pp. 586–592 (cit. on p. 61).

[52] Djemel Ziou and Madjid Allili. "Generating cubical complexes from image data and computation of the Euler number". In: *Pattern Recognition* 35.12 (2002). Pattern Recognition in Information Systems, pp. 2833–2839. ISSN: 0031-3203. DOI: https://doi.org/10.1016/S0031-3203(01)00238-2. URL: https://www.sciencedirect.com/science/article/pii/S0031320301002382 (cit. on p. 88).

[53] Nicolas Burrus et al. "A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming". In: *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*. Anaheim, CA, USA, Oct. 2003 (cit. on p. 71).

[54] ISO. *ISO/IEC 14882:2003: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Oct. 2003, p. 757. URL: https://www.iso.org/standard/38110.html (cit. on p. 70).

[55] Alexander Stepanov. *Greatest Common Measure: The Last 2500 Years*. Slides available at http://stepanovpapers.com/gcd.pdf. Lecture available on youtube at https://youtu.be/fanm5y00joc. 2003. URL: http://stepanovpapers.com/gcd.pdf (cit. on p. 57).

[56] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. Tech. rep. 2003 (cit. on p. 67).

[57] Jérôme Darbon, Thierry Géraud, and Patrick Bellot. "Generic algorithmic blocks dedicated to image processing". In: *Proceedings of the ECOOP Workshop for PhD Students*. Oslo, Norway, June 2004 (cit. on p. 72).

[58] Jacques Froment. *MegaWave2 user's guide*. 2004 (cit. on p. 50).

[59] Anat Levin, Dani Lischinski, and Yair Weiss. "Colorization Using Optimization". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. Los Angeles, California: Association for Computing Machinery, 2004, pp. 689–694. ISBN: 9781450378239. DOI: 10.1145/1186562.1015780. URL: https://doi.org/10.1145/1186562.1015780 (cit. on p. 112).

[60] Stewart Taylor. *Intel integrated performance primitives*. Intel Press, 2004 (cit. on p. 51).

[61] Valérie De Witte et al. "Vector Morphological Operators for Colour Images". In: *Image Analysis and Recognition*. Ed. by Mohamed Kamel and Aurélio Campilho. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 667–675. ISBN: 978-3-540-31938-2 (cit. on p. 62).

[62] Guntram Berti. "GrAL–the Grid Algorithms Library". In: *Future Generation Computer Systems* 22.1-2 (2006), pp. 110–122 (cit. on pp. 51, 61).

[63] Lubomir Bourdev and Hailin Jin. *Boost Generic Image Library*. Adobe stlab. Available at https://stlab.adobe.com/gil/index.html. 2006 (cit. on pp. 51, 61).

[64] Gabriel Dos Reis and Bjarne Stroustrup. "Specifying C++ Concepts". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: Association for Computing Machinery, 2006, pp. 295–308. ISBN: 1595930272. DOI: 10.1145/1111037.1111064. URL: https://doi.org/10.1145/1111037.1111064 (cit. on p. 77).

[65] Thierry Géraud. *Advanced Static Object-Oriented Programming Features: A Sequel to SCOOP*. Available at http://www.lrde.epita.fr/people/theo/pub/olena/olena-06-jan.pdf. Jan. 2006 (cit. on p. 71).

[66] Douglas Gregor et al. "Concepts: Linguistic Support for Generic Programming in C++". In: *SIGPLAN Not.* 41.10 (Oct. 2006), pp. 291–310. ISSN: 0362-1340. DOI: 10.1145/1167515.1167499. URL: https://doi.org/10.1145/1167515.1167499 (cit. on p. 56).

[67] Douglas Gregor et al. "Concepts: Linguistic Support for Generic Programming in C++". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 291–310. ISBN: 1595933484. DOI: 10.1145/1167473.1167499. URL: https://doi.org/10.1145/1167473.1167499 (cit. on p. 56).

[68] Jaakko Järvi et al. "Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++". In: *SIGPLAN Not.* 41.6 (June 2006), pp. 272–282. ISSN: 0362-1340. DOI: 10.1145/1133255.1134014. URL: http://doi.acm.org/10.1145/1133255.1134014 (cit. on p. 88).

[69] Travis Oliphant. *NumPy: A guide to NumPy*. USA: Trelgol Publishing. 2006. URL: http://www.numpy.org/ (cit. on p. 49).

[70] Jeremy Siek and Dave Abrahams. *Boost Concept Check*. https://www.boost.org/. Available starting with Boost v1.19.0. 2006 (cit. on p. 71).

[71] Jesús Angulo. "Morphological colour operators in totally ordered lattices based on distances: Application to image filtering, enhancement and analysis". In: *Computer Vision and Image Understanding* 107.1 (2007). Special issue on color image processing, pp. 56–73. ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2006.11.008. URL: https://www.sciencedirect.com/science/article/pii/S1077314206002165 (cit. on p. 62).

[72] Jeremy Gibbons. "Datatype-Generic Programming". In: *Datatype-Generic Programming: International Spring School, SSDGP 2006, Nottingham, UK, April 24-27, 2006, Revised Lectures* 4719 (2007), p. 1 (cit. on pp. 11, 26, 61).

[73] ISO. *ISO/IEC JTC 1/SC 22: Programming languages — Ada — Amendment 1*. Geneva, Switzerland: International Organization for Standardization, Mar. 2007, p. 317. URL: https://www.iso.org/standard/45001.html (cit. on p. 66).

[74] Kenneth E. Iverson. "Notation as a Tool of Thought". In: *ACM Turing Award Lectures*. New York, NY, USA: Association for Computing Machinery, 2007, p. 1979. ISBN: 9781450310499. URL: https://doi.org/10.1145/1283920.1283935 (cit. on p. 87).

[75] Travis E. Oliphant. "Multidimensional Iterators in NumPy". In: *Beautiful code*. Ed. by Andrew Oram and Greg Wilson. O'reilly Sebastopol, CA, 2007. Chap. 19 (cit. on p. 49).

[76] Bjarne Stroustrup. "Evolving a Language in and for the Real World: C++ 1991-2006". In: *Proc. of the 3rd ACM SIGPLAN Conf. on History of Programming Languages*. Vol. 4. San Diego, California, 2007, pp. 1–59. ISBN: 9781595937667. DOI: 10.1145/1238844.1238848. URL: https://doi.org/10.1145/1238844.1238848 (cit. on p. 72).

[77] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492 (cit. on pp. 56, 91).

[78] Thierry Géraud and Roland Levillain. "Semantics-Driven Genericity: A Sequel to the Static C++ Object-Oriented Programming Paradigm (SCOOP 2)". In: *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*. Paphos, Cyprus, July 2008 (cit. on p. 71).

[79] Thomas Kotzmann et al. "Design of the Java HotSpot™ Client Compiler for Java 6". In: *ACM Trans. Archit. Code Optim.* 5.1 (May 2008). ISSN: 1544-3566. DOI: 10.1145/1369396.1370017. URL: https://doi.org/10.1145/1369396.1370017 (cit. on p. 131).

[80] Roland Levillain, Thierry Géraud, and Laurent Najman. "Milena: Write Generic Morphological Algorithms Once, Run on Many Kinds of Images". In: *Mathematical Morphology and Its Application to Signal and Image Processing — Proceedings of the Ninth International Symposium on Mathematical Morphology (ISMM)*. Ed. by Michael H. F. Wilkinson and Jos B. T. M. Roerdink. Vol. 5720. Lecture Notes in Computer Science. Groningen, The Netherlands: Springer Berlin / Heidelberg, Aug. 2009, pp. 295–306 (cit. on pp. 13, 28, 51, 61, 72, 105).

[81] Fernand Meyer and Jean Stawiaski. "Morphology on graphs and minimum spanning trees". In: *Proc. of the Intl. Symp. on Mathematical Morphology (ISMM)*. Vol. 5720. LNCS. Springer, 2009, pp. 161–170 (cit. on p. 88).

[82] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, June 2009 (cit. on pp. 56, 57, 59, 72, 74).

[83] Thierry Géraud, Hugues Talbot, and Marc Van Droogenbroeck. "Algorithms for Mathematical Morphology". In: *Mathematical Morphology—From Theory to Applications*. Ed. by Laurent Najman and Hugues Talbot. ISTE. Wiley-ISTE, July 2010, pp. 323–353. ISBN: 978-1-84821-215-2. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1848212151.html (cit. on pp. 8, 23, 50, 52).

[84] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. Tuxfamily. Available at http://eigen.tuxfamily.org. 2010 (cit. on pp. 13, 28, 51, 55, 105).

[85] Roland Levillain, Thierry Géraud, and Laurent Najman. "Why and How to Design a Generic and Efficient Image Processing Framework: The Case of the Milena Library". In: *Proceedings of the IEEE International Conference on Image Processing (ICIP)*. Hong Kong, Sept. 2010, pp. 1941–1944 (cit. on pp. 13, 28, 51, 61, 72, 74, 119).

[86] Roland Levillain, Thierry Géraud, and Laurent Najman. "Writing Reusable Digital Geometry Algorithms in a Generic Image Processing Framework". In: *Proceedings of the Workshop on Applications of Digital Geometry and Mathematical Morphology (WADGMM)*. Istanbul, Turkey, Aug. 2010, pp. 96–100. URL: http://mdigest.jrc.ec.europa.eu/wadgmm2010/ (cit. on p. 72).

[87] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. "Design and evaluation of C++ open multi-methods". In: *Science of Computer Programming* 75.7 (2010). Generative Programming and Component Engineering (GPCE 2007), pp. 638–667. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2009.06.002. URL: http://www.sciencedirect.com/science/article/pii/S016764230900094X (cit. on p. 133).

[88] S. Behnel et al. "Cython: The Best of Both Worlds". In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118 (cit. on pp. 18, 34, 78, 147, 150).

[89] Lubomir Bourdev and Jaakko Järvi. "Efficient runtime dispatching in generic programming with minimal code bloat". In: *Science of Computer Programming* 76.4 (2011). Special issue on library-centric software design (LCSD 2006), pp. 243–257. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2008.06.003. URL: http://www.sciencedirect.com/science/article/pii/S0167642308000634 (cit. on p. 135).

[90] ISO. *ISO/IEC 14882:2011: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Sept. 2011, p. 1338. URL: https://www.iso.org/standard/50372.html (cit. on pp. 56, 67, 72, 73).

[91] P Kobalicek. *asmjit-complete x86/x64 JIT assembler for C++ language*. 2011 (cit. on pp. 17, 33, 147).

[92] Roland Levillain. "Towards a Software Architecture for Generic Image Processing". PhD thesis. Marne-la-Vallée, France: Université Paris-Est, Nov. 2011 (cit. on pp. 11, 25, 51, 61, 71, 72).

[93] Roland Levillain, Thierry Géraud, and Laurent Najman. "Une approche générique du logiciel pour le traitement d'images préservant les performances". In: *Proceedings of the 23rd Symposium on Signal and Image Processing (GRETSI)*. In French. Bordeaux, France, Sept. 2011 (cit. on p. 72).

[94] Jacques Froment. "MegaWave". In: *IPOL 2012 Meeting on Image Processing Libraries*. France, June 2012. URL: https://hal.archives-ouvertes.fr/hal-00907378 (cit. on p. 50).

[95] Thierry Géraud. "Outil logiciel pour le traitement d'images: Bibliothèque, paradigmes, types et algorithmes". In French. Habilitation Thesis. Université Paris-Est, June 2012 (cit. on pp. 13, 28, 51, 61, 72, 105).

[96] Thierry Géraud, Roland Levillain, and Guillaume Lazzara. *The Milena Image Processing Library*. IPOL meeting, ENS Cachan, France. June 2012. URL: https://www.lrde.epita.fr/~theo/talks/geraud.2012.ipol_talk.pdf (cit. on pp. 13, 28, 51, 61).

[97] Klaus Iglberger. *Blaze C++ Linear Algebra Library*. https://bitbucket.org/blaze-lib. 2012 (cit. on pp. 51, 55).

[98] Klaus Iglberger et al. "Expression Templates Revisited: A Performance Analysis of Current Methodologies". In: *SIAM Journal on Scientific Computing* 34(2) (2012), pp. C42–C69 (cit. on pp. 51, 55).

[99] Klaus Iglberger et al. "High Performance Smart Expression Template Math Libraries". In: *Proceedings of the 2nd International Workshop on New Algorithms and Programming Models for the Manycore Era (APMM 2012) at HPCS 2012*. 2012 (cit. on p. 55).

[100] ISO. *ISO/IEC JTC 1/SC 22: Programming languages — Ada*. Geneva, Switzerland: International Organization for Standardization, Dec. 2012, p. 832. URL: https://www.iso.org/standard/61507.html (cit. on p. 66).

[101] Roland Levillain, Thierry Géraud, and Laurent Najman. "Writing Reusable Digital Topology Algorithms in a Generic Image Processing Framework". In: *WADGMM 2010*. Ed. by Ullrich Köthe, Annick Montanvert, and Pierre Soille. Vol. 7346. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2012, pp. 140–153 (cit. on p. 72).

[102] S. Matuska, R. Hudec, and M. Benco. "The comparison of CPU time consumption for image processing algorithm in Matlab and OpenCV". In: *2012 ELEKTRO*. May 2012, pp. 75–78. DOI: 10.1109/ELEKTRO.2012.6225575 (cit. on p. 64).

[103] Bjarne Stroustrup et al. "A Concept Design for the STL". In: *C++ Standards Committee Papers, Technical Report N* 3351 (2012), pp. 12–0041 (cit. on p. 57).

[104] Andrew Sutton and Bjarne Stroustrup. "Design of Concept Libraries for C++". In: *Software Language Engineering*. Ed. by Anthony Sloane and Uwe Aßmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 97–118. ISBN: 978-3-642-28830-2 (cit. on p. 57).

[105] David Tschumperlé. "The CImg Library". In: *IPOL 2012 Meeting on Image Processing Libraries*. Cachan, France, June 2012, 4 pp. URL: https://hal.archives-ouvertes.fr/hal-00927458 (cit. on pp. 51, 61).

[106] V. Vassilev et al. "Cling — The New Interactive Interpreter for ROOT 6". In: vol. 396. 5. IOP Publishing, Dec. 2012, p. 052071. DOI: 10.1088/1742-6596/396/5/052071. URL: https://iopscience.iop.org/article/10.1088/1742-6596/396/5/052071/pdf (cit. on p. 77).

[107] AMD. *ACML-GPU - drop-in BLAS replacement, multi-GPUs accelerated*. AMD. Available at http://developer.amd.com/wordpress/media/2013/02/ACML-GPUreadme.pdf. 2013 (cit. on p. 55).

[108] Akim Demaille et al. "Implementation Concepts in Vaucanson 2". In: *Implementation and Application of Automata*. Ed. by Stavros Konstantinidis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 122–133. ISBN: 978-3-642-39274-0 (cit. on pp. 18, 33, 78, 147).

[109] Agner Fog. *C++ vector class library*. Available at http://www.agner.org/optimize/vectorclass.pdf. 2013 (cit. on p. 93).

[110] Hans J. Johnson et al. *The ITK Software Guide*. Third. *In press*. Kitware, Inc. 2013. URL: http://www.itk.org/ItkSoftwareGuide.pdf (cit. on pp. 51, 61).

[111] L. Najman and H. Talbot. *Mathematical Morphology: From Theory to Applications*. ISTE. Wiley, 2013. ISBN: 9781118600856. URL: https://books.google.fr/books?id=9FUlX8YrRvMC (cit. on pp. 8, 23, 50, 52).

[112] Jonathan Ragan-kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *PLDI 2013* (2013) (cit. on pp. 51, 55, 91, 93).

[113] Edwin Carlinet and Thierry Géraud. "A Comparative Review of Component Tree Computation Algorithms". In: *IEEE Transactions on Image Processing* 23.9 (2014), pp. 3885–3895 (cit. on p. 90).

[114] Pierre Estérie et al. "Boost.SIMD: Generic Programming for Portable SIMDization". In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '14. Orlando, Florida, USA: ACM, 2014, pp. 1–8. ISBN: 978-1-4503-2653-7. DOI: 10.1145/2568058.2568063. URL: http://doi.acm.org/10.1145/2568058.2568063 (cit. on p. 93).

[115] Matthieu Garrigues and Antoine Manzanera. "Video++, a modern image and video processing C++ framework". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2014, pp. 1–6 (cit. on p. 51).

[116] ISO. *ISO/IEC 14882:2014: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Dec. 2014, p. 1358. URL: https://www.iso.org/standard/64029.html (cit. on p. 72).

[117] Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014 (cit. on p. 57).

[118] Roland Levillain et al. "Practical Genericity: Writing Image Processing Algorithms Both Reusable and Efficient". In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications — Proceedings of the 19th Iberoamerican Congress on Pattern Recognition (CIARP)*. Ed. by Eduardo Bayro and Edwin Hancock. Vol. 8827. Lecture Notes in Computer Science. Puerto Vallarta, Mexico: Springer-Verlag, Nov. 2014, pp. 70–79 (cit. on pp. 11, 26, 51, 59, 61, 72, 74).

[119] Alexander A Stepanov and Daniel E Rose. *From mathematics to generic programming*. Pearson Education, 2014 (cit. on p. 57).

[120] Stéfan van der Walt et al. "Scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: https://doi.org/10.7717/peerj.453 (cit. on pp. 8, 22, 51, 61).

[121] Edwin Carlinet and Thierry Géraud. "MToS: A Tree of Shapes for Multivariate Images". In: *IEEE Transactions on Image Processing* 24.12 (2015), pp. 5330–5342 (cit. on p. 90).

[122] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras (cit. on p. 91).

[123] Serge Guelton et al. "Pythran: Enabling static optimization of scientific Python programs". In: *Computational Science & Discovery* 8.1 (2015), p. 014001 (cit. on pp. 18, 34, 147, 150).

[124] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 91).

[125] D Coeurjolly, JO Lachaud, and B Kerautret. *DGtal: Digital geometry tools and algorithms library*. 2016 (cit. on pp. 51, 61).

[126] Abderrahmane Nassim Halli. "Optimisation de code pour application Java haute-performance". Available at https://tel.archives-ouvertes.fr/tel-01679740/file/HALLI_2016_archivage.pdf. Theses. Université Grenoble Alpes, Oct. 2016. URL: https://tel.archives-ouvertes.fr/tel-01679740 (cit. on p. 131).

[127] Gábor Horváth and Norbert Pataki. "Transparent functors for the C++ Standard Template Library". In: *Proceedings of the 11th Joint Conference on Mathematics and Computer Science, ed. by E. Vatai, CEUR-WS*. 2016, pp. 96–101 (cit. on p. 139).

[128] ISO. *ISO/IEC JTC 1/SC 22: Programming languages — Ada — Technical Corrigendum 1*. Geneva, Switzerland: International Organization for Standardization, Feb. 2016, p. 75. URL: https://www.iso.org/standard/69798.html (cit. on p. 66).

[129] Thomas Kluyver et al. "Jupyter Notebooks — a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by Fernando Loizides and Birgit Scmidt. Netherlands: IOS Press, 2016, pp. 87–90. URL: https://eprints.soton.ac.uk/403913/ (cit. on pp. 51, 79).

[130] W. T. L. P. Lavrijsen and A. Dutta. "High-Performance Python-C++ Bindings with PyPy and Cling". In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. Nov. 2016, pp. 27–35. DOI: 10.1109/PyHPC.2016.008 (cit. on pp. 18, 34, 79, 147, 150).

[131] Conrad Sanderson and Ryan Curtin. *Armadillo, C++ library for linear algebra & scientific computing*. Sourceforge. Available at https://gitlab.com/conradsnicta/armadillo-code. 2016 (cit. on pp. 51, 55).

[132] Conrad Sanderson and Ryan Curtin. "Armadillo: a template-based C++ library for linear algebra". In: *Journal of Open Source Software* 1.2 (2016), p. 26 (cit. on p. 55).

[133] Richard Zhang, Phillip Isola, and Alexei A. Efros. "Colorful Image Colorization". In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 649–666. ISBN: 978-3-319-46487-9 (cit. on p. 112).

[134] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017 (cit. on p. 91).

[135] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV] (cit. on pp. 7, 21, 47).

[136] ISO. *ISO/IEC 14882:2017: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Dec. 2017, p. 1605. URL: https://www.iso.org/standard/68564.html (cit. on p. 72).

[137] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11—Seamless operability between C++ 11 and Python*. https://github.com/pybind/pybind11. 2017 (cit. on p. 134).

[138] *Background Subtraction*. Available at https://docs.opencv.org/3.4/d1/dc5/tutorial_background_subtraction.html. Dec. 2018 (cit. on p. 120).

[139] G. Brown et al. "Introducing Parallelism to the Ranges TS". In: *Proceedings of the International Workshop on OpenCL*. 2018, pp. 1–5 (cit. on pp. 16, 31, 125).

[140] Edwin Carlinet et al. *Pylena: a Modern C++ Image Processing Generic Library*. EPITA Research and Developement Laboratory. Available at https://gitlab.lrde.epita.fr/olena/pylene. 2018 (cit. on pp. 8, 22, 52, 60, 61, 72).

[141] Pierre Fernique and Christophe Pradal. "AutoWIG: automatic generation of Python bindings for C++ libraries". In: *PeerJ Computer Science* 4 (2018), e149. URL: https://peerj.com/articles/cs-149.pdf (cit. on pp. 18, 34, 79, 147, 150).

[142] Thierry Géraud and Edwin Carlinet. *A Modern C++ Library for Generic and Efficient Image Processing*. Journée du Groupe de Travail de Géométrie Discrète et Morphologie Mathématique, Lyon, France. June 2018. URL: https://www.lrde.epita.fr/~theo/talks/geraud.2018.gtgdmm_talk.pdf (cit. on p. 111).

[143] Eric Niebler and Casey Carter. *P1037R0: Deep Integration of the Ranges TS*. Available at https://wg21.link/p1037r0. May 2018 (cit. on pp. 9, 23, 52, 96).

[144] Sean Parent. *2018: Generic Programming*. Lecture available on youtube at https://youtu.be/FtZEU9zv9eM. 2018. URL: https://codedive.pl/2018/generic-programming (cit. on p. 56).

[145] Conrad Sanderson and Ryan Curtin. "A User-Friendly Hybrid Sparse Matrix Class in C++". In: *Mathematical Software – ICMS 2018*. Ed. by James H. Davenport et al. Cham: Springer International Publishing, 2018, pp. 422–430. ISBN: 978-3-319-96418-8 (cit. on p. 55).

[146] Michael Wong and Hal Finkel. "Distributed & Heterogeneous Programming in C++ for HPC at SC17". In: *Proceedings of the International Workshop on OpenCL*. IWOCL '18. Oxford, United Kingdom: ACM, 2018, 20:1–20:7. ISBN: 978-1-4503-6439-3. DOI: 10.1145/3204919.3204939. URL: http://doi.acm.org/10.1145/3204919.3204939 (cit. on pp. 51, 55, 93, 132).

[147] Gordon Brown, Ruyman Reyes, and Michael Wong. "Towards Heterogeneous and Distributed Computing in C++". In: *Proceedings of the International Workshop on OpenCL*. IWOCL'19. Boston, MA, USA: ACM, 2019, 18:1–18:5. ISBN: 978-1-4503-6230-6. DOI: 10.1145/3318170.3318196. URL: http://doi.acm.org/10.1145/3318170.3318196 (cit. on pp. 51, 55, 93, 132).

[148] Ben Dean. *Identifying Monoids: Exploiting Compositional Structure in Code*. Slides available at https://github.com/boostcon/cppnow_presentations_2019/blob/master/05-09-2019_thursday/Identifying_Monoids_Exploiting_Compositional_Structure_in_Code__Ben_Deane_cppnow_05092019.pdf, Lecture available on youtube at https://youtu.be/INnattuluiM. June 2019. URL: https://cppnow2019.sched.com/event/b60160aa659270370279c5acb6196fb6 (cit. on pp. 56, 94).

[149] Celian Gossec. “Binding a high-performance C++ image processing library to Python”. In: *Student LRDE tech reports*. 2019 (cit. on p. 134).

[150] Khronos Group. *OpenVX*. Available at https://www.khronos.org/openvx/. 2019 (cit. on pp. 16, 31, 125).

[151] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 91).

[152] Jules Penuchot. *CUDA extension for Blaze*. Github. Available at https://github.com/STEllAR-GROUP/blaze_cuda. 2019 (cit. on p. 55).

[153] B. Perret et al. “Higra: Hierarchical Graph Analysis”. In: *SoftwareX* 10 (2019), p. 100335. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2019.100335. URL: http://www.sciencedirect.com/science/article/pii/S235271101930247X (cit. on pp. 61, 90).

[154] PTC. *GraphicsMagick*. Version 6.0. Oct. 1, 2019. URL: https://www.mathcad.com (cit. on p. 51).

[155] Michaël Roynard, Edwin Carlinet, and Thierry Géraud. “An Image Processing Library in Modern C++: Getting Simplicity and Efficiency with Generic Programming”. In: *Reproducible Research in Pattern Recognition*. Ed. by Bertrand Kerautret et al. Cham: Springer International Publishing, 2019, pp. 121–137. ISBN: 978-3-030-23987-9 (cit. on pp. 60, 61, 171).

[156] The GIMP Development Team. *GIMP*. Version 2.10.12. June 12, 2019. URL: https://www.gimp.org (cit. on pp. 49, 50).

[157] Aksel Alpay and Vincent Heuveline. “SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL”. In: *Proceedings of the International Workshop on OpenCL*. IWOCL ’20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388658. URL: https://doi.org/10.1145/3388333.3388658 (cit. on p. 93).

[158] Anaconda, Inc. *Anaconda*. Version 2020.11. Nov. 19, 2020. URL: https://anaconda.com (cit. on p. 51).

[159] L. Thomas van Binsbergen et al. “A Principled Approach to REPL Interpreters”. In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 84–100. ISBN: 9781450381789. DOI: 10.1145/3426428.3426917. URL: https://doi.org/10.1145/3426428.3426917 (cit. on p. 130).

[160] GraphicsMagick Group. *GraphicsMagick*. Version 1.3.36. Dec. 26, 2020. URL: https://http://www.graphicsmagick.org (cit. on p. 50).

[161] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2 (cit. on pp. 19, 35, 49, 132).

[162] ISO. *ISO/IEC 14882:2020: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Dec. 2020, p. 1853. URL: https://www.iso.org/standard/79358.html (cit. on p. 96).

[163] MathWorks. *MATLAB*. Version R2020b. Sept. 22, 2020. URL: https://fr.mathworks.com/products/matlab.html (cit. on p. 51).

[164] Minh Ôn Vũ Ngoc et al. “A New Minimum Barrier Distance for Multivariate Images with Applications to Salient Object Detection, Shortest Path Finding, and Segmentation”. In: *Computer Vision and Image Understanding* 197–198 (Aug. 2020). DOI: 10.1016/j.cviu.2020.102993 (cit. on p. 88).

[165] Scilab Enterprises. *Scilab*. Version 6.1.0. Feb. 25, 2020. URL: https://www.scilab.org (cit. on p. 51).

[166] wolfram Research. *Mathematica*. Version 12.2. Dec. 16, 2020. URL: https://www.wolfram.com/mathematica/ (cit. on p. 51).

[167] Adobe. *Adobe Photoshop*. Version 22.2. Feb. 9, 2021. URL: https://photoshop.com/fr (cit. on pp. 49, 50).

[168] Alex Clark and al. *Pillow*. Version 8.1.2. Mar. 6, 2021. URL: https://python-pillow.org (cit. on p. 51).

[169] Boost. *Boost C++ Libraries*. Version 1.76.0. Apr. 16, 2021. URL: https://www.boost.org/ (cit. on pp. 8, 22, 51).

[170] GNU Project. *Octave*. Version 6.2.0. Feb. 20, 2021. URL: https://www.gnu.org/software/octave/index (cit. on p. 51).

[171] Daniel Lemire, Geoff Langdale, and John Keiser. *simdjson*. Version 1.0. Sept. 8, 2021. URL: https://github.com/simdjson/simdjson (cit. on pp. 8, 22, 51).

[172] Niels Lohmann. *Json*. Version 3.10.2. Aug. 26, 2021. URL: https://github.com/nlohmann/json (cit. on pp. 8, 22, 51).

[173] QuantStack. *xeus-cling*. Version 0.12.1. Mar. 16, 2021. URL: https://github.com/QuantStack/xeus-cling (cit. on pp. 18, 34, 79, 147, 150).

[174] The ImageMagick Development Team. *ImageMagick*. Version 7.0.10. Jan. 4, 2021. URL: https://imagemagick.org (cit. on p. 50).

[175] The PyPi Developement Team. *PyPi*. Version 21.0.1. Jan. 31, 2021. URL: https://pypi.org (cit. on p. 51).

[176] S. Behnel et al. *Using C++ in Cython > templates*. https://cython.readthedocs.io/en/latest. Available at https://cython.readthedocs.io/en/latest/src/userguide/wrapping_CPlusPlus.html#templates. 2022 (cit. on pp. 18, 34, 147).

[177] Gaël Guennebaud, Benoît Jacob, et al. *Using Eigen in CUDA kernels*. Tuxfamily. Available at https://eigen.tuxfamily.org/dox/TopicCUDA.html. 2022 (cit. on p. 55).

[178] Nvidia. *NVBLAS - drop-in BLAS replacement, multi-GPUs accelerated*. Nvidia. Available at https://docs.nvidia.com/cuda/nvblas/. 2022 (cit. on p. 55).

[179] Michaël Roynard, Edwin Carlinet, and Thierry Géraud. “A Modern C++ Point of *View* of Programming in Image Processing”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. Auckland, New Zealand, Dec. 2022. ISBN: 978-1-4503-9920-3. DOI: 10.1145/3564719.3568692 (cit. on pp. 106, 171).

[180] Codeplay. *ComputeCpp*. Version 2.8.0. URL: https://codeplay.com/products/computesuite/computecpp (cit. on p. 93).

[181] Intel Corporation. *SYCL Compiler and Runtimes*. Version master. URL: https://github.com/intel/llvm (cit. on p. 93).

[182] Danke Xie, ChernHsi Jeffrey Wang, and Michael Schimpf. “Improving Java Virtual Machine performance with SIMD technology”. In: () (cit. on p. 131).

[183] Xilinx et al. *The triSYCL project*. Version master. URL: https://github.com/trisycl/trisycl (cit. on p. 93).

[184] Andrew Sutton Eric Niebler Sean Parent. *Ranges for the Standard Library: Revision 1*. Oct. 2014. URL: https://ericniebler.github.io/std/wg21/D4128.html (cit. on pp. 13, 28, 96, 105).

[185] Bjarne Stroustrup and Gabriel Dos Reis. *Concepts — Design choices for template argument checking*. WG21, Oct. 2003. URL: https://wg21.link/n1522 (cit. on pp. 72, 73).

[186] Bill Seymour. *LWG Papers to Re-Merge into C++0x After Removing Concepts*. WG21, July 2009. URL: https://wg21.link/n2929 (cit. on pp. 56, 73).

[187] *Making Operator Functors greater<>*. WG21, Sept. 2012. URL: http://wg21.link/n3421 (cit. on p. 139).

[188] Joaquín M^a López Muñoz. *Adding heterogeneous comparison lookup to associative containers for TR2 (Rev 2)*. WG21, Oct. 2012. URL: http://wg21.link/n3465 (cit. on p. 139).

[189] Jonathan Wakely, Stephan T. Lavavej, and Joaquín M^a López Muñoz. *Adding heterogeneous comparison lookup to associative containers (rev 4)*. WG21, Mar. 2013. URL: http://wg21.link/n3657 (cit. on p. 139).

[190] Andrew Sutton. *Working Draft, C++ extensions for Concepts*. WG21, June 2017. URL: https://wg21.link/n4674 (cit. on p. 73).

[191] EPITA Research and Developpement Laboratory (LRDE). *The Olena image processing platform*. Available at http://olena.lrde.epita.fr. 2000 (cit. on pp. 51, 61, 72, 78).

[192] Ville Voutilainen. *Merge the Concepts TS Working Draft into the C++20 working draft*. WG21, June 2017. URL: https://wg21.link/p0724r0 (cit. on p. 73).

[193] Eric Niebler and Casey Carter. *Merging the Ranges TS*. WG21, May 2018. URL: https://wg21.link/p0896r1 (cit. on pp. 86, 96).

[194] Casey Carter and Eric Niebler. *Standard Library Concepts*. WG21, June 2018. URL: https://wg21.link/p0898r3 (cit. on p. 86).

[195] Eric Niebler and Casey Carter. *Deep Integration of the Ranges TS*. WG21, May 2018. URL: https://wg21.link/p1037r0 (cit. on pp. 86, 96).

[196] Bjarne Stroustrup. *Don't add to the signed/unsigned mess*. WG21, Feb. 2019. URL: https://wg21.link/p1491r0 (cit. on p. 99).

Appendix B

List of Publications

Conference Papers

Michaël Roynard, Edwin Carlinet, and Thierry Géraud. "An Image Processing Library in Modern C++: Getting Simplicity and Efficiency with Generic Programming". In: *Reproducible Research in Pattern Recognition*. Ed. by Bertrand Kerautret et al. Cham: Springer International Publishing, 2019, pp. 121–137. ISBN: 978-3-030-23987-9

Michaël Roynard, Edwin Carlinet, and Thierry Géraud. "A Modern C++ Point of *View* of Programming in Image Processing". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. Auckland, New Zealand, Dec. 2022. ISBN: 978-1-4503-9920-3. DOI: 10.1145/3564719.3568692

Appendix C

Concepts & archetypes

C.1 The fundamentals

C.1.1 Value

Concept table The details of the concept *Value* are presented in appendix C.1.1.

Concept	Modeling type	Inherit behavior from	Instance of type
Value	`Val`	∅	`val`
ComparableValue	`CmpVal`	Value	`cmp_val1, cmp_val2`
OrderedValue	`OrdVal`	ComparableValue	`ord_val1, ord_val2`

Concept	Expression	Return Type	Description
Value	`std::semiregular<Val>`	`std::true_type`	`Val` is a semiregular type. It can be: copied, moved, swapped, and default constructed.
ComparableValue	`std::regular<CmpVal>`	`std::true_type`	`CmpVal` is a regular type. It is a semiregular type that is equality comparable.
	`cmp_val1 == cmp_val2`	`boolean`	Supports equality comparison
OrderedValue	`std::totally_ordered<OrdVal>`	`std::true_type`	`CmpVal` is a totally ordered as well as a regular type. Additionally the expressions must be equality preserving.
	`ord_val1 < ord_val2`	`boolean`	Supports inequality comparisons
	`ord_val1 <= ord_val2, ...`	`boolean`	

Table C.1: Concepts Value: expressions

Concept code

```
// Value
template <typename Val>
concept Value = std::semiregular<Val>;

// ComparableValue
template <typename RegVal>
concept ComparableValue =
  std::regular<RegVal>;

// OrderedValue
template <typename STORegVal>
concept OrderedValue =
  std::regular<STORegVal> &&
  std::totally_ordered<STORegVal>;
```

Archetype code

```
struct Value
{
};
```

```
struct ComparableValue
{
};
bool operator==(const ComparableValue&, const ComparableValue&);
bool operator!=(const ComparableValue&, const ComparableValue&);


struct OrderedValue
{
};
bool operator==(const OrderedValue&, const OrderedValue&);
bool operator!=(const OrderedValue&, const OrderedValue&);
bool operator<(const OrderedValue&, const OrderedValue&);
bool operator>(const OrderedValue&, const OrderedValue&);
bool operator<=(const OrderedValue&, const OrderedValue&);
bool operator>=(const OrderedValue&, const OrderedValue&);

static_assert(mln::concepts::Value<Value>,
    "Value archetype does not model the Value concept!");
static_assert(mln::concepts::ComparableValue<ComparableValue>,
    "ComparableValue archetype does not model the ComparableValue concept!");
static_assert(mln::concepts::OrderedValue<OrderedValue>,
    "OrderedValue archetype does not model the OrderedValue concept!");
```

C.1.2 Point

Concept table The details of the concept *Point* are presented in appendix C.1.2.

Concept	Modeling type	Inherit behavior from	Instance of type
Point	`Pnt`	∅	`pnt1, pnt2`

Concept	Expression	Return Type	Description
Point	`std::regular<Pnt>`	`std::true_type`	`Pnt` is a regular type. It can be: copied, moved, swapped, and default constructed. It also is equality comparable.
	`std::totally_ordered<OrdVal>`	`std::true_type`	`Pnt` is a totally ordered as well as a regular type. Additionally the expressions must be equality preserving.
	`pnt1 < pnt2` `pnt1 <= pnt2, ...`	`boolean` `boolean`	supports inequality comparisons

Table C.2: Concepts Point: expressions

Concept code

```
// Point
template <typename P>
concept Point =
  std::regular<P> &&
  std::totally_ordered<P>;
```

Archetype code

```
struct Point final
{
};

bool operator==(const Point&, const Point&);
bool operator!=(const Point&, const Point&);
bool operator<(const Point&, const Point&);
bool operator>(const Point&, const Point&);
bool operator<=(const Point&, const Point&);
bool operator>=(const Point&, const Point&);

static_assert(mln::concepts::Point<Point>, "Point archetype does not model the Point concept!");
```

C.1.3 Pixel

Concept table The details of the concept *Pixel* are presented in appendix C.1.3 and appendix C.1.3.

Concept	Modeling type	Inherit behavior from	Instance of type
Pixel	`Pix`	∅	`pix`
OutputPixel	`OPix`	Pixel	`opix`

Concept	Definition	Description	Requirement
Pixel	`value_type`	Type of the value contained in the pixel. Cannot be constant or reference.	Models the concept `Value`.
	`reference_type`	Type used to mutate the pixel's value if non-const. Can be a proxy.	Models the concept `std::indirectly_writable` if non-const.
	`point_type`	Type of the pixel's point.	Models the concept `Point`

Table C.3: Concepts Pixel: definitions

Type	Instance of type
`Pix::value_type`	`val`
`Pix::point_type`	`pnt`

Concept	Expression	Return Type	Description
Pixel	`pix.val()`	`Pix::reference_type`	Access the pixel's value for read and/or write purpose.
	`pix.point()`	`Pix::point_type`	Read the pixel's point.
	`pix.shift(pnt)`	`void`	Shift pixel's point coordinate base on `pnt`'s coordinates.
OutputPixel	`opix.val() = val`	`void`	Mutate pixel's value.

Table C.4: Concepts Pixel: expressions

Concept code

```
// Pixel
template <class Pix>
concept Pixel =
  std::is_base_of_v<mln::details::Pixel<Pix>, Pix> &&
  std::copy_constructible<Pix> &&
  std::move_constructible<Pix> &&
  requires {
    typename pixel_value_t<Pix>;
    typename pixel_reference_t<Pix>;
    typename pixel_point_t<Pix>;
  } &&
  std::semiregular<pixel_value_t<Pix>> &&
  Point<pixel_point_t<Pix>> &&
  !std::is_const_v<pixel_value_t<Pix>> &&
  !std::is_reference_v<pixel_value_t<Pix>> &&
  requires(const Pix cpix, Pix pix, pixel_point_t<Pix> p) {
    { cpix.point() } -> std::convertible_to<pixel_point_t<Pix>>;
    { cpix.val() }   -> std::convertible_to<pixel_reference_t<Pix>>;
    { pix.shift(p) };
  };

// WritablePixel
template <typename WPix>
concept WritablePixel =
  Pixel<WPix> &&
  requires(const WPix cpix, pixel_value_t<WPix> v) {
    // Not deep-const, view-semantic.
    { cpix.val() = v };
    // Proxy rvalues must not be deep-const on their assignement semantic (unlike tuple...)
    { const_cast<typename WPix::reference const &&>(cpix.val()) = v };
  };

// OutputPixel
template <typename Pix>
concept OutputPixel = detail::WritablePixel<Pix>;
```

Archetype code

```
namespace details
{
  template <class P, class V>
  struct PixelT
  {
    using value_type = V;
    using point_type = P;
    using reference  = const value_type&;

    PixelT()              = delete;
    PixelT(const PixelT&) = default;
    PixelT(PixelT&&)      = default;
    PixelT& operator=(const PixelT&) = delete;
    PixelT& operator=(PixelT&&) = delete;

    point_type point() const;
    reference  val() const;
    void       shift(const P& dp);
  };

  struct OutputPixel : PixelT<Point, Value>
  {
    using reference = Value&;
    reference val() const;
  };


  template <class Pix>
  struct AsPixel : Pix, mln::details::Pixel<AsPixel<Pix>>
  {
  };
} // namespace details

template <class P, class V = Value>
using PixelT      = details::AsPixel<details::PixelT<P, V>>;
using Pixel       = PixelT<Point, Value>;
using OutputPixel = details::AsPixel<details::OutputPixel>;

static_assert(mln::concepts::Pixel<Pixel>,
    "Pixel archetype does not model the Pixel concept!");
static_assert(mln::concepts::OutputPixel<OutputPixel>,
    "OutputPixel archetype does not model the OutputPixel concept!");
```

C.1.4 Ranges

Concept table The details of the concept *Ranges* are presented in appendix C.1.4 and appendix C.1.4.

Concept	Modeling type	Inherit behavior from	Instance of type
MDRange	`MDRng`	∅	`mdrng`
OutputMDRange	`OMDRng`	MDRange	`omdrng`
ReversibleMDRange	`RMDRng`	MDRange	`rmdrng`

Concept	Definition	Description	Requirement
MDRange	`value_type`	Type of the value contained in the range. Cannot be constant or reference.	Models the concept `Value`.
	`reference_type`	Type used to mutate the pixel's value if non-const. Can be a proxy.	Models the concept `std::indirectly_writable` if non-const.

Table C.5: Concepts Ranges: definitions

Type	Instance of type
`std::ranges::range_value_t<MDRng>`	`val`

Concept	Expression	Return Type	Description
MDRange	`mdrng.begin()`	`unspecified`	Return a forward iterator allowing a traversing of the range.
	`mdrng.end()`	`unspecified`	Return a sentinel allowing to know when the end is reached.
OutputMDRange	`auto it = omdrng.begin()` `*it++ = val`	`void`	Mutate a value inside the range then increment the iterator's position
ReversibleMDRange	`rmdrng.rbegin()`	`unspecified`	Return a forward iterator allowing a traversing of the range starting from the end.
	`rmdrng.rend()`	`unspecified`	Return a sentinel allowing to know when the end is reached.

Table C.6: Concepts Ranges: expressions

Concept code

```
template <class C>
concept MDCursor =
  std::ranges::detail::forward_cursor<C> &&
  std::ranges::detail::forward_cursor<std::ranges::detail::begin_cursor_t<C>> &&
  requires (C c)
  {
    { c.read() } -> std::ranges::forward_range;
    c.end_cursor();
  };

template <class C>
concept NDCursor = std::semiregular<C> &&
  requires (C c)
  {
    { C::rank } -> std::same_as<int>;
    c.read();
    c.move_to_next(0);
    c.move_to_end(0);
  };

template <class C>
concept MDBidirectionalCursor = MDCursor<C> &&
  requires (C c)
  {
    c.move_to_prev();
    c.move_to_prev_line();
  };
```

```
template <class R>
concept MDRange =
  requires(R r)
  {
    { r.rows() } -> std::ranges::forward_range;
    { r.begin_cursor() } -> MDCursor;
    { r.end_cursor() } -> std::same_as<std::ranges::default_sentinel_t>;
  };

template <class R>
concept MDBidirectionalRange = MDRange<R> &&
  requires (R r)
  {
    { r.rrows() } -> std::ranges::forward_range;
    { r.rbegin_cursor() } -> MDCursor;
    { r.rend_cursor() } -> std::same_as<std::ranges::default_sentinel_t>;
  };

template <class R>
concept mdrange = MDRange<R> || std::ranges::range<R>;

template <class R, class V>
concept output_mdrange = mdrange<R> && std::ranges::output_range<mdrange_row_t<R>, V>;

template <class R>
concept reversible_mdrange = MDBidirectionalRange<R> || std::ranges::bidirectional_range<R>;
```

C.1.5 Domain

Concept table The details of the concept *Domain* are presented in table C.7 and appendix C.1.5.

Concept	Modeling type	Inherit behavior from	Instance of type
Domain	`MDRng`	MDRange	`dom`
SizedDomain	`OMDRng`	Domain	`sdom`
ShapedDomain	`RMDRng`	SizedDomain	`shdom`

Table C.7: Concepts Domain: definitions

Type	Instance of type
`Dom::value_type`	`pnt`

Concept	Expression	Return Type	Description
Domain	`Point<Dom::value_type>`	`std::true_type`	Domain's value models the `Point` concept
	`dom.has(pnt)`	`bool`	Check if a points is included in the domain.
	`dom.empty()`	`void`	Read the pixel's point.
	`dom.dim()`	`void`	Returns the domain's dimension.
SizedDomain	`sdom.size()`	`unsigned int`	Returns the number of points inside the domain.
ShapedDomain	`shdom.extends()`	`std::forward_range`	Return a range that yields the number of elements for each dimension.

Table C.8: Concepts Domain: expressions

Concept code

```
// Domain
template <typename Dom>
concept Domain =
  mln::ranges::mdrange<Dom> &&
  Point<mln::ranges::mdrange_value_t<Dom>> &&
  requires(const Dom cdom, mln::ranges::mdrange_value_t<Dom> p) {
    { cdom.has(p) }   -> std::same_as<bool>;
    { cdom.empty() }  -> std::same_as<bool>;
    { cdom.dim() }    -> std::same_as<int>;
  };


// SizedDomain
template <typename Dom>
concept SizedDomain =
  Domain<Dom> &&
  requires(const Dom cdom) {
    { cdom.size() } -> std::unsigned_integral;
  };

// ShapedDomain
template <typename Dom>
concept ShapedDomain =
  SizedDomain<Dom> &&
  requires(const Dom cdom) {
    { cdom.extents() }  -> std::ranges::forward_range;
  };
```

Archetype code

```
struct Domain
{
  using value_type = Point;
  using reference  = Point&;

  value_type* begin();
```

```
  value_type* end();

  bool has(value_type) const;
  bool empty() const;
  int  dim() const;
};

static_assert(mln::concepts::Domain<Domain>,
    "Domain archetype does not model the Domain concept!");

struct SizedDomain : Domain
{
  unsigned size() const;
};

static_assert(mln::concepts::SizedDomain<SizedDomain>,
                          "SizedDomain archetype does not model the SizedDomain concept!");

struct ShapedDomain final : SizedDomain
{
  static constexpr std::size_t  ndim = 1;
  value_type                    shape() const;
  std::array<std::size_t, ndim> extents() const;
};

static_assert(mln::concepts::ShapedDomain<ShapedDomain>,
                          "ShapedDomain archetype does not model the ShapedDomain concept!");
```

C.1.6 Image

Concept table The details of the concept *Image* are presented in table C.9 and appendix C.1.6.

Concept	Modeling type	Inherit behavior from	Instance of type
Image (InputImage, ForwardImage)	`Img`	∅	`img`
WritableImage	`WImg`	Image	`wimg`

Concept	Definition	Description	Requirement
	`pixel_type`	Type of the image's pixel.	Models the concept `Pixel`.
	`point_type`	Type of the image's point.	Models the concept `Point`.
	`value_type`	Type of the image's value. Cannot be constant or reference	Models the concept `Value`.
	`domain_type`	Type of the image's domain.	Models the concept `Domain`.
	`reference`	Type used to mutate an image pixel's value if non-const	Models the concept `std::indirectly_writable` if non-const.
	`concrete_type`	Image concrete type (that holds data).	Models the concept `Image`.
	`ch_value_type<V>`	Facility to return a new image type. that casts the underlying `value_type` into `V`	Models the concept `Image`.

Table C.9: Concepts Image: definitions (1)

Concept	Expression	Return Type	Description
Image	`img.concretize()`	`std::convertible_to< concrete_type>`	Return a concrete image that holds data.
	`img.ch_value<V>()`	`std::convertible_to< ch_value_type<V> >`	Return an image whose values are casted to `V`.
	`img.domain()`	`std::convertible_to< domain_type>`	Return the image's domain.
	`img.pixels()`	`MDRange`	Return a range that yields all the image pixels.
	`img.values()`		Return a range that yields all the image values.
	`std::convertible_to< std::ranges::ranges_value_t< decltype(img.pixels())>, pixel_type>` `std::convertible_to< std::ranges::ranges_value_t< decltype(img.values())>, value_type>`	`std::true_type`	Ranges converts to compatible element types.
WritableImage	`wimg.values()`	`OutputMDRange`	Return a range that yields all the image values (mutable).
	`OutputPixel< std::ranges::ranges_value_t< decltype(wimg.pixels())> >`	`std::true_type`	Ranges whose elements are mutable.

Table C.10: Concepts Image: expressions (1)

Concept code

```
template <typename I>
concept Image =
  // Minimum constraint on image object
  // Do not requires DefaultConstructible
  std::is_base_of_v<mln::details::Image<I>, I> &&
  std::copy_constructible<I> &&
  std::move_constructible<I> &&
  std::derived_from<image_category_t<I>, forward_image_tag> &&
  requires {
    typename image_pixel_t<I>;
    typename image_point_t<I>;
    typename image_value_t<I>;
```

```
    typename image_domain_t<I>;
    typename image_reference_t<I>;
    typename image_concrete_t<I>;
    typename image_ch_value_t<I, mln::archetypes::Value>;
    // traits
    typename image_indexable_t<I>;
    typename image_accessible_t<I>;
    typename image_extension_category_t<I>;
    typename image_category_t<I>;
    typename image_view_t<I>;
  } &&
  Pixel<image_pixel_t<I>> &&
  Point<image_point_t<I>> &&
  Value<image_value_t<I>> &&
  Domain<image_domain_t<I>> &&
  std::convertible_to<pixel_point_t<image_pixel_t<I>>, image_point_t<I>> &&
  std::convertible_to<pixel_reference_t<image_pixel_t<I>>, image_reference_t<I>> &&
  // Here we don't want a convertible constraint as value_type
  // is the decayed type and should really be the same
  std::same_as<pixel_value_t<image_pixel_t<I>>, image_value_t<I>> &&
  std::common_reference_with<image_reference_t<I>&&, image_value_t<I>&> &&
  std::common_reference_with<image_reference_t<I>&&, image_value_t<I>&&> &&
  std::common_reference_with<image_value_t<I>&&, const image_value_t<I>&> &&
  requires(I ima, const I cima, image_domain_t<I> d, image_point_t<I> p) {
    { cima.template ch_value<mln::archetypes::Value>() }
        -> std::convertible_to<image_ch_value_t<I, mln::archetypes::Value>>;
    { cima.concretize() } -> std::convertible_to<image_concrete_t<I>>;
    { cima.domain() }     -> std::convertible_to<image_domain_t<I>>;
    { ima.pixels() }  -> mln::ranges::mdrange;
    { ima.values() }  -> mln::ranges::mdrange;
    requires std::convertible_to<mln::ranges::mdrange_value_t<decltype(ima.pixels())>,
                                 image_pixel_t<I>>;
    requires std::convertible_to<mln::ranges::mdrange_value_t<decltype(ima.values())>,
                                 image_value_t<I>>;
  };

namespace detail
{
  // WritableImage
  template <typename I>
  concept WritableImage =
    Image<I> &&
    OutputPixel<image_pixel_t<I>> &&
    requires(I ima) {
    { ima.values() }  -> mln::ranges::output_mdrange<image_value_t<I>>;
      // Check Writability of each pixel of the range
      requires OutputPixel<
                  std::common_type_t<
                    mln::ranges::mdrange_value_t<decltype(ima.pixels())>,
                    image_pixel_t<I>>>;
    };
} // namespace detail


// InputImage
template <typename I>
concept InputImage = Image<I>;


// ForwardImage
template <typename I>
concept ForwardImage = InputImage<I>;
```

Archetype code

```
namespace details
{
  template <class I>
  struct AsImage : I, mln::details::Image<AsImage<I>>
  {
    using I::I;

    using concrete_type = AsImage<typename I::concrete_type>;
```

```
  concrete_type concretize() const;


  template <typename V>
  using ch_value_type = AsImage<typename I::template ch_value_type<V>>;

  template <typename V>
  ch_value_type<V> ch_value() const;
};


struct Image
{
  using pixel_type = archetypes::Pixel;
  using value_type     = pixel_value_t<mln::archetypes::Pixel>;
  using reference      = pixel_reference_t<mln::archetypes::Pixel>;
  using point_type     =  std::ranges::range_value_t<Domain>;
  using domain_type    = Domain;
  using category_type  = forward_image_tag;
  using concrete_type  = Image;

  template <class V>
  using ch_value_type = Image;

  // additional traits
  using extension_category = mln::extension::none_extension_tag;
  using indexable          = std::false_type;
  using accessible         = std::false_type;
  using view               = std::false_type;

  Image()                     = default;
  Image(const Image&) = default;
  Image(Image&&)      = default;
  Image& operator=(const Image&) = default;
  Image& operator=(Image&&) = default;

  domain_type domain() const;


  struct pixel_range
  {
    const pixel_type* begin();
    const pixel_type* end();
  };
  pixel_range pixels();


  struct value_range
  {
    const value_type* begin();
    const value_type* end();
  };

  value_range values();
};

struct OutputImage : Image
{
  using pixel_type = archetypes::OutputPixel;
  using reference  = pixel_reference_t<mln::archetypes::OutputPixel>;

  struct pixel_range
  {
    const pixel_type* begin();
    const pixel_type* end();
  };

  pixel_range pixels();

  struct value_range
  {
    value_type* begin();
    value_type* end();
```

```
    };

    value_range values();
  };
} // namespace details


using Image         = details::AsImage<details::Image>;
using ForwardImage  = Image;
using WritableImage = details::AsImage<details::OutputImage>;
```

C.2 Advanced way to access image data

C.2.1 Index

Concept table The details of the concept *Index* are presented in appendix C.2.1.

Concept	Modeling type	Inherit behavior from	Instance of type
Index	`Idx`	∅	`idx, idy`

Concept	Expression	Return Type	Description
Index	`std::signed_integral<Idx>`	`std::true_type`	Idx is a signed integral arithmetic type
	`idx + idy, idx - idy, ...`	`Idx`	Supports all trivial arithmetical operations

Table C.11: Concepts Index: expressions

Concept code

```
// Index
template <typename Idx>
concept Index = std::signed_integral<Idx>;
```

Archetype code

```
using Index = int;

static_assert(mln::concepts::Index<Index>, "Index archetype does not model the Index concept!");
```

C.2.2 Indexable image

Concept table The details of the concept *Indexable image* are presented in table C.12 and appendix C.2.2.

Concept	Modeling type	Inherit behavior from	Instance of type
IndexableImage	`IdxImg`	Image	`idximg`
WritableIndexableImage	`WIdxImg`	IndexableImage, WritableImage	`widximg`

Concept	Definition	Description	Requirement
IndexableImage	`index_type`	Type of the image's buffer index.	Models the concept `Index`.

Table C.12: Concepts Image: definitions (2)

Type	Instance of type
`Img::value_type`	`val`
`IdxImg::index_type`	`k`

Concept	Expression	Return Type	Description
IndexableImage	`idximg[k]`	`std::same_as<reference>`	Access a value at a given index.
Writable IndexableImage	`widximg[k] = val`	`void`	Mutate a value at a given index.

Table C.13: Concepts Image: expressions (2)

Concept code

```
// IndexableImage
template <typename I>
concept IndexableImage =
  Image<I> &&
  requires {
    typename image_index_t<I>;
  } &&
  image_indexable_v<I> &&
  requires (I ima, image_index_t<I> k) {
    { ima[k] }  -> std::same_as<image_reference_t<I>>;  // For concrete image it returns
                                                        // a const_reference
  };


namespace detail
{
  // WritableIndexableImage
  template <typename I>
  concept WritableIndexableImage =
    WritableImage<I> &&
    IndexableImage<I> &&
    requires(I ima, image_index_t<I> k, image_value_t<I> v) {
      { ima[k] = v }
    };
} // namespace detail
```

Archetype code

```
namespace details
{

  struct IndexableImage : Image
  {
    using index_type = int;
    using indexable  = std::true_type;

    using concrete_type = IndexableImage;
```

```
    template <class V>
    using ch_value_type = IndexableImage;


    reference operator[](index_type);
  };

  struct OutputIndexableImage : OutputImage
  {
    using index_type = int;
    using indexable  = std::true_type;

    using concrete_type = OutputIndexableImage;

    template <class V>
    using ch_value_type = OutputIndexableImage;

    reference operator[](index_type);
  };

} // namespace details

using IndexableImage              = details::AsImage<details::IndexableImage>;
using WritableIndexableImage      = details::AsImage<details::OutputIndexableImage>;
```

C.2.3 Accessible image

Concept table The details of the concept *Accessible image* are presented in table C.14 and appendix C.2.3.

Concept	Modeling type	Inherit behavior from	Instance of type
AccessibleImage	`AccImg`	Image	`accimg`
WritableAccessibleImage	`WAccImg`	AccessibleImage, WritableImage	`waccimg`

Table C.14: Concepts Image: definitions (3)

Type	Instance of type
`Img::point_type`	`pnt`

Concept	Expression	Return Type	Description
AccessibleImage	`accimg(pnt)`	`std::same_as<reference>`	Access a value for a given point.
	`accimg.at(pnt)`	`std::same_as<reference>`	Access a value for a given point. No bound checking.
	`accimg.pixel(pnt)`	`std::same_as<pixel_type>`	Access a pixel for a given point.
	`accimg.pixel_at(pnt)`	`std::same_as<pixel_type>`	Access a pixel for a given point. No bound checking.
Writable AccessibleImage	`img(pnt) = val`	`void`	Mutate a value at a given point.
	`waccimg.at(pnt) = val`	`void`	Mutate a value at a given point. No bound checking.
	`OutputPixel<decltype( waccimg.pixel(pnt))>`	`std::true_type`	The returned pixel models `OutputPixel`.
	`OutputPixel<decltype( waccimg.pixel_at(pnt))>`	`std::true_type`	The returned pixel models `OutputPixel`. No bound checking.

Table C.15: Concepts Image: expressions (3)

Concept code

```
// AccessibleImage
template <typename I>
concept AccessibleImage =
  Image<I> &&
  image_accessible_v<I> &&
  requires (I ima, image_point_t<I> p) {
    { ima(p) }              -> std::same_as<image_reference_t<I>>; // For concrete image it returns
                                                                   // a const_reference
    { ima.at(p) }           -> std::same_as<image_reference_t<I>>; // idem
    { ima.pixel(p) }    -> std::same_as<image_pixel_t<I>>; // For concrete image pixel may propagate
                                                           // constness
    { ima.pixel_at(p) } -> std::same_as<image_pixel_t<I>>; // idem
  };


namespace detail
{
  // WritableAccessibleImage
  template <typename I>
  concept WritableAccessibleImage =
    detail::WritableImage<I> &&
    AccessibleImage<I> &&
    requires(I ima, image_point_t<I> p, image_value_t<I> v) {
      { ima(p) = v };
      { ima.at(p) = v };

      requires OutputPixel<decltype(ima.pixel(p))>;
      requires OutputPixel<decltype(ima.pixel_at(p))>;
    };
} // namespace detail
```

Archetype code

```
namespace details {
  struct OutputAccessibleImage : OutputImage
  {
    using accessible    = std::true_type;
    using concrete_type = OutputAccessibleImage;

    template <class V>
    using ch_value_type = OutputAccessibleImage;


    reference      operator()(point_type);
    reference      at(point_type);
    pixel_type pixel(point_type);
    pixel_type pixel_at(point_type);
  };


  struct AccessibleImage : Image
  {
    using accessible    = std::true_type;
    using concrete_type = AccessibleImage;

    template <class V>
    using ch_value_type = AccessibleImage;

    reference      operator()(point_type);
    reference      at(point_type);
    pixel_type pixel(point_type);
    pixel_type pixel_at(point_type);
  };
} // namespace details

using AccessibleImage             =
        details::AsImage<details::AccessibleImage>;
using WritableAccessibleImage     =
        details::AsImage<details::OutputAccessibleImage>;
```

C.2.4 Indexable and accessible image

Concept table The details of the concept *Indexable and accessible image* are presented in table C.16 and appendix C.2.4.

Concept	Modeling type	Inherit behavior from	Instance of type
IndexableAnd AccessibleImage	`IdxAccImg`	IndexableImage, AccessibleImage	`idxaccimg`
WritableIndexable AndAccessibleImage	`WIdxAccImg`	IndexableAndAccessibleImage, WritableIndexableImage, WritableAccessibleImage	`widxaccimg`

Table C.16: Concepts Image: definitions (4)

Concept	Expression	Return Type	Description
IndexableAnd AccessibleImage	`img.point_at_index(k)`	`point_type`	Get the point corresponding to the given index.
	`idxaccimg.index_of_point(pnt)`	`index_type`	Get the linear index (offset in the buffer) of multi-dimensional point.
	`idxaccimg.delta_index(pnt)`	`index_type`	Get the linear index offset for the given point.

Table C.17: Concepts Image: expressions (4)

Concept code

```
// IndexableAndAccessibleImage
template <typename I>
concept IndexableAndAccessibleImage =
  IndexableImage<I> &&
  AccessibleImage<I> &&
  requires (const I cima, image_index_t<I> k, image_point_t<I> p) {
    { cima.point_at_index(k) }  -> std::same_as<image_point_t<I>>;
    { cima.index_of_point(p) }  -> std::same_as<image_index_t<I>>;
    { cima.delta_index(p) }     -> std::same_as<image_index_t<I>>;
  };


namespace detail
{
  // WritableIndexableAndAccessibleImage
  template <typename I>
  concept WritableIndexableAndAccessibleImage =
    IndexableAndAccessibleImage<I> &&
    detail::WritableImage<I> &&
    detail::WritableIndexableImage<I>;
} // namespace detail
```

Archetype code

```
namespace details {
  struct OutputIndexableAndAccessibleImage : OutputAccessibleImage
  {
    using index_type = int;
    using indexable  = std::true_type;

    using concrete_type = OutputIndexableAndAccessibleImage;

    template <class V>
    using ch_value_type = OutputIndexableAndAccessibleImage;


    reference  operator[](index_type);
    point_type point_at_index(index_type) const;
    index_type index_of_point(point_type) const;
```

```
    index_type delta_index(point_type) const;
  };

  struct IndexableAndAccessibleImage : AccessibleImage
  {
    using index_type = int;
    using indexable  = std::true_type;

    using concrete_type = IndexableAndAccessibleImage;

    template <class V>
    using ch_value_type = IndexableAndAccessibleImage;

    reference  operator[](index_type);
    point_type point_at_index(index_type) const;
    index_type index_of_point(point_type) const;
    index_type delta_index(point_type) const;
  };
} // namespace details

using IndexableAndAccessibleImage       = details::AsImage<details::IndexableAndAccessibleImage>;
using WritableIndexableAndAccessibleImage = details::AsImage<details::OutputIndexableAndAccessibleImage>;
```

C.2.5 Bidirectional image

Concept table The details of the concept *Bidirectional image* are presented in table C.18 and table C.18.

Concept	Modeling type	Inherit behavior from	Instance of type
BidirectionalImage	`BidirImg`	Image	`bidirimg`
WritableBidirectionalImage	`WBidirImg`	BidirectionalImage, WritableImage	`wbidirimg`

Table C.18: Concepts Image: definitions (5)

Type	Instance of type
`Img::point_type`	`pnt`
`Img::value_type`	`val`
`IdxImg::index_type`	`k`
`int`	`dim`

Concept	Expression	Return Type	Description
BidirectionalImage	`bidirimg.pixels()`	`ReversibleMDRange`	Return a reversible range that yields all the image pixels.
	`bidirimg.values()`		Return a reversible range that yields all the image values.

Table C.19: Concepts Image: expressions (5)

Concept code

```
namespace detail
{
  // WritableBidirectionalImage
  template <typename I>
  concept WritableBidirectionalImage =
    WritableImage<I> &&
    BidirectionalImage<I>;
} // namespace detail

// RawImage (not contiguous, stride = padding)
template <typename I>
concept RawImage =
  IndexableAndAccessibleImage<I> &&
  BidirectionalImage<I> &&
  std::derived_from<image_category_t<I>, raw_image_tag> &&
  requires (I ima, const I cima, int dim) {
    { ima.data() }        -> std::convertible_to<const image_value_t<I>*>; // data() may be proxied
                                                                           // by a view
    { cima.stride(dim) } -> std::same_as<std::ptrdiff_t>;
  };
```

Archetype code

```
namespace details {
  struct BidirectionalImage : Image
  {
    using category_type = bidirectional_image_tag;

    struct pixel_range
    {
      const pixel_type* begin();
      const pixel_type* end();
      pixel_range          reversed();
    };

    pixel_range pixels();

    struct value_range
```

```
    {
      const value_type* begin();
      const value_type* end();
      value_range       reversed();
    };

    value_range values();
  };

  struct OutputBidirectionalImage : BidirectionalImage
  {
    using pixel_type = archetypes::OutputPixel;
    using reference      = pixel_reference_t<mln::archetypes::OutputPixel>;

    struct value_range
    {
      value_type* begin();
      value_type* end();
      value_range reversed();
    };
    value_range values();


    struct pixel_range
    {
      const pixel_type* begin();
      const pixel_type* end();
      pixel_range           reversed();
    };

    pixel_range pixels();
  };
} // namespace details

using BidirectionalImage = details::AsImage<details::BidirectionalImage>;
using OutputBidirectionalImage = details::AsImage<details::OutputBidirectionalImage>;
```

C.2.6 Raw image

Concept table The details of the concept *Raw image* are presented in table C.20 and table C.20.

Concept	Modeling type	Inherit behavior from	Instance of type
RawImage	`RawImg`	IndexableAndAccessibleImage, BidirectionalImage	`rawimg`
WritableRawImage	`WRawImg`	RawImage, WritableIndexableImage, WritableBidirectionalImage	`wrawimg`

Table C.20: Concepts Image: definitions (6)

Concept	Expression	Return Type	Description
RawImage	`rawimg.data()`	`std::convertible_to<const value_type*>`	Get a constant pointer to the first element of the domain.
	`rawimg.stride(dim)`	`std::ptrdiff_t`	Get the stride (in number of elements) between two consecutive elements in the given `dim`.
Writable RawImage	`img.data()`	`std::convertible_to<value_type*>`	Get a pointer to the first element of the domain.
	`*(wrawimg.data() + k) = val`	`void`	Access an element from the data buffer at index `k` and mutate it to `val`.
WithExtension Image	`wextimg.extension()`	`std::convertible_to<extension_type>`	Get the extension of the image.

Table C.21: Concepts Image: expressions (6)

Concept code

```
// RawImage (not contiguous, stride = padding)
template <typename I>
concept RawImage =
  IndexableAndAccessibleImage<I> &&
  BidirectionalImage<I> &&
  std::derived_from<image_category_t<I>, raw_image_tag> &&
  requires (I ima, const I cima, int dim) {
    { ima.data() }        -> std::convertible_to<const image_value_t<I>*>; // data() may be proxied
                                                                           // by a view
    { cima.stride(dim) } -> std::same_as<std::ptrdiff_t>;
  };


namespace detail
{
  // WritableRawImage
  template <typename I>
  concept WritableRawImage =
    WritableIndexableAndAccessibleImage<I> &&
    WritableBidirectionalImage<I> &&
    RawImage<I> &&
    requires(I ima, image_value_t<I> v, image_index_t<I> k) {
      { ima.data() }        -> ::concepts::convertible_to<image_value_t<I>*>;
      { *(ima.data() + k) = v };
    };
} // namespace detail
```

Archetype code

```
namespace details {
  struct RawImage : IndexableAndAccessibleImage
  {
    using category_type = raw_image_tag;
    using pixel_range   = BidirectionalImage::pixel_range;
    using value_range   = BidirectionalImage::value_range;
```

```
    pixel_range pixels();
    value_range values();


    const value_type* data() const;
    std::ptrdiff_t    strides(int) const;
  };

  struct OutputRawImage : OutputIndexableAndAccessibleImage
  {
    using category_type = raw_image_tag;
    using pixel_range   = OutputBidirectionalImage::pixel_range;
    using value_range   = OutputBidirectionalImage::value_range;

    pixel_range pixels();
    value_range values();

    value_type*    data() const;
    std::ptrdiff_t strides(int) const;
  };
} // namespace details

using RawImage          = details::AsImage<details::RawImage>;
using OutputRawImage    = details::AsImage<details::OutputRawImage>;
```

C.3 Local algorithm concepts: structuring elements and extensions

C.3.1 Structuring element

Concept table The details of the concept *Structuring element* are presented in table C.22 and appendix C.3.1.

Concept	Modeling type	Inherit behavior from	Instance of type
StructuringElement	`SE, Pnt`	∅, Point	`se, pnt`
DecomposableStructuringElement	`DSE, Pnt`	StructuringElement, Point	`dse`
SeparableStructuringElement	`SSE, Pnt`	StructuringElement, Point	`sse`
IncrementalStructuringElement	`ISE, Pnt`	StructuringElement, Point	`ise`

Concept	Definition	Description	Requirement

StructuringElement	`incremental`
	`decomposable`
	`separable`
		`std::bool_constant`	`std::true_type` if supported
			`std::false_false` otherwise.

Table C.22: Concepts Structuring Elements: definitions

Concept code

```
namespace details
{
  template <typename SE>
  concept DynamicStructuringElement =
    requires (SE se) {
    { se.radial_extent() }  -> std::same_as<int>;
    };

  constexpr bool implies(bool a, bool b) { return !a || b; }
}

template <typename SE, typename P>
concept StructuringElement =
  std::convertible_to<SE, mln::details::StructuringElement<SE>> &&
  std::ranges::regular_invocable<SE, P> &&
  std::ranges::regular_invocable<SE, mln::archetypes::PixelT<P>> &&
  requires {
    typename SE::category;
    typename SE::incremental;
    typename SE::decomposable;
    typename SE::separable;
  } &&
  std::convertible_to<typename SE::category, mln::adaptative_neighborhood_tag> &&
  details::implies(std::convertible_to<typename SE::category, mln::dynamic_neighborhood_tag>,
                   details::DynamicStructuringElement<SE>) &&
  requires (SE se, const SE cse, P p, mln::archetypes::PixelT<P> px) {
    { se(p) }         -> std::ranges::forward_range;
    { se(px) }        -> std::ranges::forward_range;
    { cse.offsets() } -> std::ranges::forward_range;

    requires std::convertible_to<std::ranges::range_value_t<decltype(se(p))>, P>;
    requires std::Pixel<std::ranges::range_value_t<decltype(se(px))>>;
    requires std::convertible_to<std::ranges::range_value_t<decltype(cse.offsets())>, P>;
  };

namespace details
{
  template <typename R, typename P>
  concept RangeOfStructuringElement =
    StructuringElement<std::ranges::range_value_t<R>, P>;
}

template <typename SE, typename P>
concept DecomposableStructuringElement =
```

Type	Instance of type	Requirement
`Pix`	`pix`	`std::same_as<Pix::point_type, Pnt>`
`Pnt`	`pnt`	

Concept	Expression	Return Type	Description
StructuringElement	`std::regular_invocable<SE, Pnt>` `std::regular_invocable<SE, Pix>`	`std::true_type`	`se` can called using std::invoke, is equality preserving and does not modify function object nor arguments.
	`se(pnt)`	`std::forward_range`	Return a range that yeilds the neighboring points of `pnt`.
	`se.offsets()`	`std::forward_range`	Return a range that yeilds the neighboring points, in relative coordinates.
	`se(pix)`	`std::forward_range`	Return a range that yeilds the neighboring pixels of `pix`.
	`se.radial_extent()`	`int`	Returns the radial extent of the SE, the radius of the disc (square).

	`std::convertible_to<`
	`std::ranges::range_value_t<`
	`decltype(se(pnt))>, Pnt>`
	`std::convertible_to<`
	`std::ranges::range_value_t<`
	`decltype(se.offsets())>, Pnt>`
	`Pixel<std::ranges::range_value_t<`
	`decltype(se(pix))> >`
		`std::true_type`	Converts to a compatible point type
			
			Is a range of compatible pixel type

Decomposable StructuringElement	`dse.is_decomposable()`	`std::bool_constant`	`std::true_type` if supported `std::false_false` otherwise.
	`dse.decompose()`	`std::forward_range`	Return a range that yeilds simpler structuring elements.

	`StructuringElement<`
	`std::ranges::range_value_t<`
	`decltype(dse.decompose())> >`
		`std::true_type`	Is a range of compatible structuring elements types.

	`dse.is_decomposable()`	`bool`	Wether the decompose facility is supported
Separable StructuringElement	`dse.is_decomposable()`	`std::bool_constant`	`std::true_type` if supported `std::false_false` otherwise.
	`sse.separate()`	`std::forward_range`	Return a range that yeilds simpler structuring elements.

	`StructuringElement<`
	`std::ranges::range_value_t<`
	`decltype(sse.separate())> >`
		`std::true_type`	Is a range of compatible structuring elements types.

	`sse.is_separable()`	`bool`	Wether the separate facility is supported
Incremental StructuringElement	`dse.is_decomposable()`	`std::bool_constant`	`std::true_type` if supported `std::false_false` otherwise.
	`ise.inc()`	`StructuringElement<` `SE, Pnt>`	Return the next simpler structuring elements.
	`ise.dec()`	`StructuringElement<` `SE, Pnt>`	Return the previous simpler structuring elements.
	`ise.is_incremental()`	`bool`	Wether the incremental facility is supported

Table C.23: Concepts Structuring Elements: expressions

```
  StructuringElement<SE, P> &&
  std::convertible_to<typename SE::decomposable, std::true_type> &&
  requires(const SE se) {
    { se.is_decomposable() }  -> std::same_as<bool>;
    { se.decompose() }        -> std::ranges::forward_range;
    requires details::RangeOfStructuringElement<decltype(se.decompose()), P>;
  };

template <typename SE, typename P>
concept SeparableStructuringElement =
  StructuringElement<SE, P> &&
  std::convertible_to<typename SE::separable, std::true_type> &&
  requires(const SE se) {
    { se.is_separable() } -> std::same_as<bool>;
    { se.separate() }     -> std::ranges::forward_range;
    requires details::RangeOfStructuringElement<decltype(se.separate()), P>;
  };

template <typename SE, typename P>
concept IncrementalStructuringElement =
  StructuringElement<SE, P> &&
  std::convertible_to<typename SE::incremental, std::true_type> &&
  requires(const SE se) {
    { se.is_incremental() } -> std::same_as<bool>;
    { se.inc() }  -> StructuringElement<SE, P>;
    { se.dec() }  -> StructuringElement<SE, P>;
  };
```

Achetype code

```
namespace details
{
  template <class P, class Pix>
  requires mln::concepts::Point<P>&& mln::concepts::Pixel<Pix>
  struct StructuringElement
  {
    using category     = adaptative_neighborhood_tag;
    using incremental  = std::false_type;
    using decomposable = std::false_type;
    using separable    = std::false_type;

      std::ranges::subrange<P*> operator()(P p);

      std::ranges::subrange<Pix*> operator()(Pix px);
      std::ranges::subrange<P*>   offsets() const;
  };


  template <class SE>
  struct AsSE : SE, mln::details::Neighborhood
  helper<AsSE<SE>>
  {
  };
} // namespace details

template <class P = Point, class Pix = PixelT<P>>
using StructuringElement = details::AsSE<details::StructuringElement<P, Pix>>;


namespace details
{
  template <class P, class Pix>
  struct DecomposableStructuringElement : StructuringElement<P, Pix>
  {
    using decomposable = std::true_type;

    bool                                                          is_decomposable() const;
    std::ranges::subrange<mln::archetypes::StructuringElement<P, Pix>*> decompose() const;
  };

  template <class P, class Pix>
  struct SeparableStructuringElement : StructuringElement<P, Pix>
  {
```

```
    using separable = std::true_type;

    bool                                                                  is_separable() const;
    std::ranges::subrange<mln::archetypes::StructuringElement<P, Pix>*> separate() const;
  };

  template <class P, class Pix>
  struct IncrementalStructuringElement : StructuringElement<P, Pix>
  {
    using incremental = std::true_type;

    bool                                 is_incremental() const;
    archetypes::StructuringElement<P, Pix> inc()          const;
    archetypes::StructuringElement<P, Pix> dec()          const;
  };
} // namespace details


template <class P = Point, class Pix = PixelT<P>>
using DecomposableStructuringElement = details::AsSE<details::DecomposableStructuringElement<P, Pix>>;

template <class P = Point, class Pix = PixelT<P>>
using SeparableStructuringElement = details::AsSE<details::SeparableStructuringElement<P, Pix>>;

template <class P = Point, class Pix = PixelT<P>>
using IncrementalStructuringElement = details::AsSE<details::IncrementalStructuringElement<P, Pix>>;
```

C.3.2 Neighborhood

Concept table The details of the concept *Neighborhood* are presented in table C.24 and appendix C.3.2.

Concept	Modeling type	Inherit behavior from	Instance of type
Neighborhood	`SE, Pnt`	StructuringElement, Point	`se, pnt`

Table C.24: Concepts Neighborhood: definitions

Type	Instance of type	Requirement
`Pix`	`pix`	`std::same_as<Pix::point_type, Pnt>`
`Pnt`	`pnt`	

Concept	Expression	Return Type	Description
Neighborhood	`se.before(pnt)`	`std::forward_range`	Return a range that yields the points before `pnt`.
	`se.after(pnt)`		Return a range that yields the points after `pnt`.
	`se.before(pix)`		Return a range that yields the pixels before `pix`.
	`se.after(pix)`		Return a range that yields the pixels after `pnt`.
	`std::convertible_to< std::ranges::ranges_value_t< decltype(se.before(pnt))>, Pnt>`	`std::true_type`	Ranges converts to compatible element types.
	`std::convertible_to< std::ranges::ranges_value_t< decltype(se.after(pnt))>, Pnt>`		
	`Pixel< std::ranges::ranges_value_t< decltype(se.before(pix))>, Pix>`		Ranges that are of compatible element types.
	`Pixel< std::ranges::ranges_value_t< decltype(se.after(pix))>, Pix>`		

Table C.25: Concepts Neighborhood: expressions

Concept code

```
template <typename SE, typename P>
concept Neighborhood =
  StructuringElement<SE, P> &&
  requires (SE se, P p, mln::archetypes::PixelT<P> px) {
    { se.before(p) }  -> std::ranges::forward_range;
    { se.after(p) }   -> std::ranges::forward_range;
    { se.before(px) } -> std::ranges::forward_range;
    { se.after(px) }  -> std::ranges::forward_range;

    requires std::convertible_to<std::ranges::range_value_t<decltype(se.before(p))>, P>;
    requires std::convertible_to<std::ranges::range_value_t<decltype(se.after(p))>, P>;
    requires std::Pixel<std::ranges::range_value_t<decltype(se.before(px))>>;
    requires std::Pixel<std::ranges::range_value_t<decltype(se.after(px))>>;
  };
```

Archetype code

```
  namespace details
  {
    template <class P, class Pix>
    requires mln::concepts::Point<P>&& mln::concepts::Pixel<Pix>
    struct Neighborhood : StructuringElement<P, Pix>
    {
       std::ranges::iterator_range<P*>   before(P p);
       std::ranges::iterator_range<P*>   after(P p);
       std::ranges::iterator_range<Pix*> before(Pix px);
       std::ranges::iterator_range<Pix*> after(Pix px);
    };
```

```
    template <class N>
    struct AsNeighborhood : N, mln::details::Neighborhood<AsNeighborhood<N>>
    {
    };

  } // namespace details

  template <class P = Point, class Pix = PixelT<P>>
  using Neighborhood = details::AsSE<details::Neighborhood<P, Pix>>;
```

C.3.3 Extensions

Concept table The details of the concept *Extensions* are presented in table C.26 and appendix C.3.3.

Concept	Modeling type	Inherit behavior from	Instance of type
Extension	`Ext, Pnt`	∅, Point	`ext, pnt`
FillableExtension	`FExt, Pnt`	Extension, Point	`fext`
MirrorableExtension	`MExt, Pnt`	Extension, Point	`mext`
PeriodizableExtension	`PExt, Pnt`	Extension, Point	`pext`
ClampableExtension	`CExt, Pnt`	Extension, Point	`cext`
ExtendWithExtension	`EwExt, Pnt, U`	Extension, Point, Image	`ewext, u`

Concept	Definition	Description	Requirement
Extension	`value_type`	Type of value contained in the range. Cannot be constant or reference.	Models the concept `Value`.

	`support_fill`
	`support_mirror`
	`support_periodize`
	`support_clamp`
	`support_extend_with`
		`std::bool_constant`	`std::true_type` if supported
			`std::false_false` otherwise.

ExtendWithExtension	`point_type`	Type of point in the extended image.	Converts to the sub-image points type.

Table C.26: Concepts Extensions: definitions

Concept code

```
template <typename Ext, typename Pnt>
concept Extension =
  std::is_base_of_v<mln::Extension<Ext>, Ext> &&
  requires {
    typename Ext::support_fill;
    typename Ext::support_mirror;
    typename Ext::support_periodize;
    typename Ext::support_clamp;
    typename Ext::support_extend_with;
  } &&
  Value<typename Ext::value_type> &&
  requires (const Ext cext,
      mln::archetypes::StructuringElement<
        Pnt,
        mln::archetypes::Pixel> se) {
    { cext.fit(se) }      -> std::same_as<bool>;
    { cext.extent() }     -> std::same_as<int>;
  };

template <typename Ext, typename Pnt>
concept FillableExtension =
  Extension<Ext, Pnt> &&
  std::convertible_to<typename Ext::support_fill, std::true_type> &&
  requires {
    typename Ext::value_type;
  } &&
  requires (Ext ext, const Ext cext, const typename Ext::value_type& v) {
    { ext.fill(v) };
    { cext.is_fill_supported() }  -> std::same_as<bool>;
  };

template <typename Ext, typename Pnt>
concept MirrorableExtension =
  Extension<Ext, Pnt> &&
  std::convertible_to<typename Ext::support_mirror, std::true_type> &&
  requires (Ext ext, const Ext cext) {
    { ext.mirror() };
    { cext.is_mirror_supported() }  -> std::same_as<bool>;
  };
```

Type	Instance of type
`SE<Pnt, Pix>`	`se`
`Ext::value_type`	`val`
`Ext::point_type`	`offset`

Concept	Expression	Return Type	Description
Extension	`ext.fit(se)`	`bool`	Wether the extension fit the structuring element.
	`ext.extent()`	`int`	Extension's border width. std::numeric_limits<int>::max() for infinite size.
FillableExtension	`fext.fill(val)`	`void`	Fill the extension with the value `val`.
	`fext.is_fill_supported`	`bool`	Wether the fill facility is supported.
MirrorableExtension	`mext.mirror()`	`void`	Fill the extension with mirrored image's values.
	`mext.is_mirror_supported()`	`bool`	Wether the mirror facility is supported.
PeriodizeableExtension	`mext.periodize()`	`void`	Fill the extension with periodic copies of image's values.
	`mext.is_periodize_supported()`	`bool`	Wether the mirror facility is supported.
ClampableExtension	`mext.clamp()`	`void`	Fill the extension by extending image's values at extremities.
	`mext.is_clamp_supported()`	`bool`	Wether the clamp facility is supported.
ExtendWithExtension	`std::convertible_to<point_type, U::point_type>`	`std::true_type`	Converts to the sub-image points type.
	`mext.extend_with(u, offset)`	`void`	Fill the extension with sub-image `u`'s values starting from `offset`.
	`mext.is_extent_with_supported()`	`bool`	Wether the extend-with-sub-image facility is supported.

Table C.27: Concepts Extensions: expressions

```
template <typename Ext, typename Pnt>
concept PeriodizableExtension =
  Extension<Ext, Pnt> &&
  std::convertible_to<typename Ext::support_periodize, std::true_type> &&
  requires (Ext ext, const Ext cext) {
    { ext.periodize() };
    { cext.is_periodize_supported() }  -> std::same_as<bool>;
  };

template <typename Ext, typename Pnt>
concept ClampableExtension =
  Extension<Ext, Pnt> &&
  std::convertible_to<typename Ext::support_clamp, std::true_type> &&
  requires (Ext ext, const Ext cext) {
    { ext.clamp() };
    { cext.is_clamp_supported() }  -> std::same_as<bool>;
  };

template <typename Ext, typename Pnt, typename U>
concept ExtendWithExtension =
  Extension<Ext, Pnt> &&
  std::convertible_to<typename Ext::support_extend_with, std::true_type> &&
  InputImage<U> &&
  requires {
    typename Ext::point_type;
  } &&
  std::convertible_to<typename U::value_type, typename Ext::value_type> &&
  std::convertible_to<typename Ext::point_type, typename U::point_type> &&
  requires (Ext ext, const Ext cext, U u, typename Ext::point_type offset) {
    { ext.extend_with(u, offset) };
    { cext.is_extend_with_supported() }  -> std::same_as<bool>;
  };
```

C.3.4 Extended image

Concept table The details of the concept *Extended image* are presented in table C.28 and appendix C.3.4.

Concept	Modeling type	Inherit behavior from	Instance of type
WithExtensionImage	`WExtImg`	Image	`wextimg`
ConcreteImage	`CImg`	Image	`cimg`
ViewImage	`VImg`	Image	`vimg`

Concept	Definition	Description	Requirement
WithExtensionImage	`extension_type`	Type of the image's extension.	Models the concept `Extension`.

Table C.28: Concepts Image: definitions (7)

Concept	Expression	Return Type	Description
WithExtension Image	`wextimg.extension()`	`std::convertible_to< extension_type>`	Get the extension of the image.

Table C.29: Concepts Image: expressions (7)

Concept code

```
template <typename I>
concept WithExtensionImage =
  Image<I> &&
  requires {
    typename image_extension_t<I>;
  } &&
  Extension<image_extension_t<I>, image_point_t<I>> &&
  not std::same_as<mln::extension::none_extension_tag, image_extension_category_t<I>> &&
  requires (I ima, image_point_t<I> p) {
    { ima.extension() } -> std::convertible_to<image_extension_t<I>>;
  };
```

Archetype code

```
namespace details {
  struct WithExtensionImage : Image
  {
    struct Extension : ::mln::Extension<Extension>
    {
      using support_fill = std::false_type;
      using support_mirror = std::false_type;
      using support_periodize = std::false_type;
      using support_clamp = std::false_type;
      using support_extend_with = std::false_type;
      using value_type = image_value_t<Image>;
      bool fit(mln::archetypes::StructuringElement<image_point_t<Image>, mln::archetypes::Pixel> se) const;
      int extent() const;
    };

    using extension_type = Extension;

    using extension_category = mln::extension::custom_extension_tag;

    extension_type extension() const;
  };
} // namespace details

using WithExtensionImage = details::AsImage<details::WithExtensionImage>;
```

C.3.5 Output image

Concept table The details of the concept *Output image* are presented in appendix C.3.5.

$$\begin{aligned} OutputImage = &(Image \implies WritableImage) \wedge \\ &(IndexableImage \implies WritableIndexableImage) \wedge \\ &(AccessibleImage \implies WritableAccessibleImage) \wedge \\ &(IndexableAndAccessibleImage \implies WritableIndexableAndAccessibleImage) \wedge \\ &(BidirectionalImage \implies WritableBidirectionalImage) \wedge \\ &(RawImage \implies WritableRawImage) \end{aligned} \tag{C.1}$$

Figure C.1: Concepts OutputImage: definition

Concept code

```
// OutputImage
// Usage: RawImage<I> && OutputImage<I>
template <typename I>
concept OutputImage =
  (not Image<I> || (detail::WritableImage<I>)) &&
  (not IndexableImage<I> || (detail::WritableIndexableImage<I>)) &&
  (not AccessibleImage<I> || (detail::WritableAccessibleImage<I>)) &&
  (not IndexableAndAccessibleImage<I> ||
    (detail::WritableIndexableAndAccessibleImage<I>)) &&
  (not BidirectionalImage<I> || (detail::WritableBidirectionalImage<I>)) &&
  (not RawImage<I> || (detail::WritableRawImage<I>));
```

Appendix D

Static-dynamic bridge

D.1 Ndimage module

Here is the code of the ndimage module, including the helpers for the type conversion.

ndimage.hpp

```
#include <mln/core/image/ndbuffer_image.hpp>

#include <pybind11/numpy.h>
#include <pybind11/pybind11.h>


namespace mln::py
{
  mln::ndbuffer_image from_numpy(pybind11::array arr);

  pybind11::object to_numpy(const mln::ndbuffer_image& img);

  void init_pylena_numpy(pybind11::module& m);
} // namespace mln::py

namespace pybind11::detail
{
  template <>
  struct type_caster<mln::ndbuffer_image>
  {
    PYBIND11_TYPE_CASTER(mln::ndbuffer_image, _("numpy.ndarray"));

    bool load(handle h, bool)
    {
      pybind11::array arr = reinterpret_borrow<pybind11::array>(h);
      value               = mln::py::from_numpy(arr);
      return true;
    }

    static handle cast(const mln::ndbuffer_image& img, return_value_policy, handle)
    {
      return mln::py::to_numpy(img).inc_ref();
    }
  };
} // namespace pybind11::detail
```

ndimage.cpp

```
#include "ndimage.hpp"

#include <mln/core/image/ndimage.hpp>

#include <fmt/core.h>
#include <pybind11/cast.h>

#include <cassert>
```

```cpp
#include <stdexcept>
#include <string>


namespace
{

  namespace details
  {
    template <mln::sample_type_id T>
    static pybind11::dtype dtype_of()
    {
      return pybind11::dtype::of<typename mln::sample_type_id_traits<T>::type>();
    }
  } // namespace details

  pybind11::dtype get_sample_type(mln::sample_type_id type)
  {
    switch (type)
    {
    case mln::sample_type_id::INT8:
      return details::dtype_of<mln::sample_type_id::INT8>();
    case mln::sample_type_id::INT16:
      return details::dtype_of<mln::sample_type_id::INT16>();
    case mln::sample_type_id::INT32:
      return details::dtype_of<mln::sample_type_id::INT32>();
    case mln::sample_type_id::INT64:
      return details::dtype_of<mln::sample_type_id::INT64>();
    case mln::sample_type_id::UINT8:
      return details::dtype_of<mln::sample_type_id::UINT8>();
    case mln::sample_type_id::UINT16:
      return details::dtype_of<mln::sample_type_id::UINT16>();
    case mln::sample_type_id::UINT32:
      return details::dtype_of<mln::sample_type_id::UINT32>();
    case mln::sample_type_id::UINT64:
      return details::dtype_of<mln::sample_type_id::UINT64>();
    case mln::sample_type_id::FLOAT:
      return details::dtype_of<mln::sample_type_id::FLOAT>();
    case mln::sample_type_id::DOUBLE:
      return details::dtype_of<mln::sample_type_id::DOUBLE>();
    case mln::sample_type_id::BOOL:
      return details::dtype_of<mln::sample_type_id::BOOL>();
    case mln::sample_type_id::RGB8:
      return details::dtype_of<mln::sample_type_id::UINT8>();
    default:
      throw std::runtime_error("Invalid sample_type_id");
    }
    return pybind11::none();
  }

  mln::sample_type_id get_sample_type(const std::string& type_format)
  {
    pybind11::dtype type;
    try
    {
      type = pybind11::dtype(type_format);
    }
    catch (const std::exception&)
    {
      return mln::sample_type_id::OTHER;
    }
    if (type.is(details::dtype_of<mln::sample_type_id::INT8>()))
      return mln::sample_type_id::INT8;
    else if (type.is(details::dtype_of<mln::sample_type_id::INT16>()))
      return mln::sample_type_id::INT16;
    else if (type.is(details::dtype_of<mln::sample_type_id::INT32>()))
      return mln::sample_type_id::INT32;
    else if (type.is(details::dtype_of<mln::sample_type_id::INT64>()))
      return mln::sample_type_id::INT64;
    else if (type.is(details::dtype_of<mln::sample_type_id::UINT8>()))
      return mln::sample_type_id::UINT8;
    else if (type.is(details::dtype_of<mln::sample_type_id::UINT16>()))
      return mln::sample_type_id::UINT16;
```

```
    else if (type.is(details::dtype_of<mln::sample_type_id::UINT32>()))
      return mln::sample_type_id::UINT32;
    else if (type.is(details::dtype_of<mln::sample_type_id::UINT64>()))
      return mln::sample_type_id::UINT64;
    else if (type.is(details::dtype_of<mln::sample_type_id::FLOAT>()))
      return mln::sample_type_id::FLOAT;
    else if (type.is(details::dtype_of<mln::sample_type_id::DOUBLE>()))
      return mln::sample_type_id::DOUBLE;
    else if (type.is(details::dtype_of<mln::sample_type_id::BOOL>()))
      return mln::sample_type_id::BOOL;
    return mln::sample_type_id::OTHER;
  }
} // namespace


namespace mln::py
{
  mln::ndbuffer_image from_numpy(pybind11::array arr)
  {
    if (!pybind11::detail::check_flags(arr.ptr(),
    pybind11::detail::npy_api::NPY_ARRAY_C_CONTIGUOUS_))
      throw std::invalid_argument("Array should be C contiguous");
    auto                base = arr.base();
    const auto          info = arr.request();
    mln::sample_type_id type = get_sample_type(info.format);
    if (type == mln::sample_type_id::OTHER)
      throw std::invalid_argument(fmt::format(
          "Invalid dtype argument (Got dtype format {} expected types:"
          " [u]int[8, 16, 32, 64], float, double or bool)",
          info.format));
    const bool is_rgb8 = info.ndim == 3 && info.shape[2] == 3 &&
                         type == mln::sample_type_id::UINT8;
    const auto pdim    = info.ndim - (is_rgb8 ? 1 : 0);
    if (pdim > mln::PYLENE_NDBUFFER_DEFAULT_DIM)
      throw std::invalid_argument(
          fmt::format("Invalid number of dimension from numpy array"
                      " (Got {} but should be less than {})", pdim,
                      mln::PYLENE_NDBUFFER_DEFAULT_DIM));
    int            size[mln::PYLENE_NDBUFFER_DEFAULT_DIM]    = {0};
    std::ptrdiff_t strides[mln::PYLENE_NDBUFFER_DEFAULT_DIM] = {0};
    for (auto d = 0; d < pdim; d++)
    {
      size[d]    = info.shape[pdim - d - 1];
      strides[d] = info.strides[pdim - d - 1];
    }
    const auto sample_type = is_rgb8 ? mln::sample_type_id::RGB8 : type;
    auto       res =
        mln::ndbuffer_image::from_buffer(reinterpret_cast<std::byte*>(info.ptr),
                                         sample_type, pdim, size, strides);
    if (base && pybind11::isinstance<mln::internal::ndbuffer_image_data>(base))
      res.__data() = pybind11::cast<std::shared_ptr<mln::internal::ndbuffer_image_data>>(base);
    return res;
  }


  pybind11::object to_numpy(const mln::ndbuffer_image& img)
  {
    const auto&      api    = pybind11::detail::npy_api::get();
    pybind11::object data   = pybind11::none();
    int              flags = pybind11::detail::npy_api::NPY_ARRAY_WRITEABLE_;
    if (img.__data())
    {
      data = pybind11::cast(img.__data());
      assert(data.ref_count() > 0);
    }

    /* For the moment, restrict RGB8 image to 2D image */
    const bool is_rgb8 = img.pdim() == 2 && img.sample_type() == mln::sample_type_id::RGB8;
    const auto ndim    = img.pdim() + (is_rgb8 ? 1 : 0);
    std::vector<std::size_t> strides(ndim, 1);
    std::vector<std::size_t> shapes(ndim, 3);
    auto                     descr = get_sample_type(img.sample_type());
```

```
    for (auto d = 0; d < img.pdim(); d++)
    {
      strides[d] = img.byte_stride(img.pdim() - d - 1);
      shapes[d]  = img.size(img.pdim() - d - 1);
    }

    auto res = pybind11::reinterpret_steal<pybind11::object>(api.PyArray_NewFromDescr_(
        api.PyArray_Type_, descr.release().ptr(), ndim, reinterpret_cast<Py_intptr_t*>(shapes.data()),
        reinterpret_cast<Py_intptr_t*>(strides.data()),
        reinterpret_cast<void*>(img.buffer()), flags, nullptr));

    if (!res)
      throw std::runtime_error("Unable to create the numpy array in ndimage -> array");
    if (data)
      // **Steal** a reference to data
      // (https://numpy.org/devdocs/reference/c-api/array.html#c.PyArray_SetBaseObject)
      api.PyArray_SetBaseObject_(res.ptr(), data.release().ptr());
    return res;
  }

  void init_pylena_numpy(pybind11::module& m)
  {
    if (!pybind11::detail::get_global_type_info(typeid(mln::internal::ndbuffer_image_data)))
    {
      pybind11::class_<mln::internal::ndbuffer_image_data,
        std::shared_ptr<mln::internal::ndbuffer_image_data>>(
          m, "ndbuffer_image_data");
    }
  }

} // namespace mln::py
```

D.2 Structuring element module

Here is the code of the structuring element module. This module defines what structuring elements available from Python.

se.hpp

```
#include <pybind11/pybind11.h>


namespace mln::py
{

  void init_module_se(pybind11::module& m);

}
```

se.cpp

```
#include "se.hpp"

#include <mln/core/image/ndimage.hpp>
#include <mln/core/se/disc.hpp>
#include <mln/core/se/rect2d.hpp>

#include <pybind11/pybind11.h>


namespace mln::py
{

  void init_module_se(pybind11::module& m)
  {
    pybind11::class_<mln::se::disc_non_decomp>(m, "disc").def(
        pybind11::init([](float radius) { return mln::se::disc_non_decomp{radius}; }));

    pybind11::class_<mln::se::disc>(m, "disc_decomposable").def(
      pybind11::init([](float radius) {
        return mln::se::disc{radius};
    }));

    pybind11::class_<mln::se::rect2d_non_decomp>(m, "rect2d").def(
      pybind11::init([](int width, int height) {
        return mln::se::rect2d_non_decomp{width, height};
    }));

    pybind11::class_<mln::se::rect2d>(m, "rect2d_decomposable").def(
      pybind11::init([](int width, int height) {
        return mln::se::rect2d{width, height};
    }));
  }

} // namespace mln::py
```

D.3 Mathematical morphology module

This module defines the routines that are exposed and available from Python. It is divided in three sub-parts. First is the $n \times n$ dispatcher code. Second is the value set mechanic. Third is the code exposing the mathematical morphology algorithms.

D.3.1 The $n \times n$ dispatcher

visit.hpp

```
#include <mln/core/image/ndimage.hpp>

#include <stdexcept>

namespace mln::py
{
  template <template <typename> class F, typename... Args>
  auto visit(mln::sample_type_id tid, Args&&... args)
  {
    switch (tid)
    {
    case (mln::sample_type_id::INT8):
      return F<std::int8_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT16):
      return F<std::int16_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT32):
      return F<std::int32_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT64):
      return F<std::int64_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT8):
      return F<std::uint8_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT16):
      return F<std::uint16_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT32):
      return F<std::uint32_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT64):
      return F<std::uint64_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::FLOAT):
      return F<float>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::DOUBLE):
      return F<double>{}(std::forward<Args>(args)...);
    /*case (mln::sample_type_id::RGB8):
      return F<mln::rgb8>{}(std::forward<Args>(args)...);*/
    case (mln::sample_type_id::OTHER):
      [[fallthrough]];
    default:
      throw std::runtime_error("Unhandled data type");
    }
  }

  template <template <typename> class F, typename D, typename... Args>
  auto visit_d(mln::sample_type_id tid, D&& d, Args&&... args)
  {
    switch (tid)
    {
    case (mln::sample_type_id::INT8):
      return F<std::int8_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT16):
      return F<std::int16_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT32):
      return F<std::int32_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::INT64):
      return F<std::int64_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT8):
      return F<std::uint8_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT16):
      return F<std::uint16_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT32):
      return F<std::uint32_t>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::UINT64):
      return F<std::uint64_t>{}(std::forward<Args>(args)...);
```

```
    case (mln::sample_type_id::FLOAT):
      return F<float>{}(std::forward<Args>(args)...);
    case (mln::sample_type_id::DOUBLE):
      return F<double>{}(std::forward<Args>(args)...);
    /*case (mln::sample_type_id::RGB8):
      return F<mln::rgb8>{}(std::forward<Args>(args)...);*/
    case (mln::sample_type_id::OTHER):
      [[fallthrough]];
    default:
      return std::forward<D>(d);
    }
  }
} // namespace mln::py
```

D.3.2 Value-set mechanics

value_set.hpp

```
#include <algorithm>
#include <any>
#include <cassert>
#include <cmath>
#include <limits>
#include <type_traits>
#include <utility>


namespace mln
{

  template <class T = void>
  struct value_set
  {
    template <class U>
    U cast(T v) const
    {
      return static_cast<U>(v);
    }

    T max() const noexcept { return std::numeric_limits<T>::max(); }
    T min() const noexcept { return std::numeric_limits<T>::min(); }
    T sup() const noexcept { return std::numeric_limits<T>::max(); }
    T inf() const noexcept { return std::numeric_limits<T>::min(); }

    T plus(T v) const noexcept { return +v; }
    T minus(T v) const noexcept { return -v; }

    auto max(T l, T r) const noexcept { return std::max(l, r); }
    auto min(T l, T r) const noexcept { return std::min(l, r); }
    auto add(T l, T r) const noexcept { return l + r; }
    auto sub(T l, T r) const noexcept { return l - r; }
    auto mult(T l, T r) const noexcept { return l * r; }
    auto div(T l, T r) const noexcept { return l / r; }
    auto mod(T l, T r) const noexcept { return l % r; }
    auto pow(T l, T r) const noexcept { return std::pow(l, r); }
  };

  template <>
  struct value_set<void>
  {

    template <class U, class T>
    U cast(T&& t) const
    {
      return static_cast<U>(std::forward<T>);
    }

    template <class T>
    T max() const noexcept
    {
      return std::numeric_limits<T>::max();
    }
```

```
  template <class T>
  T min() const noexcept
  {
    return std::numeric_limits<T>::min();
  }
  template <class T>
  T sup() const noexcept
  {
    return std::numeric_limits<T>::max();
  }
  template <class T>
  T inf() const noexcept
  {
    return std::numeric_limits<T>::min();
  }

  template <class T>
  T plus(T&& v) const noexcept
  {
    return +std::forward<T>(v);
  }
  template <class T>
  T minus(T&& v) const noexcept
  {
    return -std::forward<T>(v);
  }

  template <class T, class U>
  auto max(T&& l, T&& r) const noexcept
  {
    return std::max(std::forward<T>(l), std::forward<T>(r));
  }
  template <class T, class U>
  auto min(T&& l, T&& r) const noexcept
  {
    return std::min(std::forward<T>(l), std::forward<T>(r));
  }

  template <class T, class U>
  auto add(T&& l, U&& r) const noexcept
  {
    return std::forward<T>(l) + std::forward<T>(r);
  }
  template <class T, class U>
  auto sub(T&& l, U&& r) const noexcept
  {
    return std::forward<T>(l) - std::forward<T>(r);
  }
  template <class T, class U>
  auto mult(T&& l, U&& r) const noexcept
  {
    return std::forward<T>(l) * std::forward<T>(r);
  }
  template <class T, class U>
  auto div(T&& l, U&& r) const noexcept
  {
    return std::forward<T>(l) / std::forward<T>(r);
  }
  template <class T, class U>
  auto mod(T&& l, U&& r) const noexcept
  {
    return std::forward<T>(l) % std::forward<T>(r);
  }
  template <class T, class U>
  auto pow(T&& l, U&& r) const noexcept
  {
    return std::pow(std::forward<T>(l), std::forward<T>(r));
  }
};

struct abstract_value_set
{
  virtual ~abstract_value_set() {}
```

```
  /*
  template <class U>
  virtual std::any cast(std::any v) const = 0;
  */

  virtual std::any max() const = 0;
  virtual std::any min() const = 0;
  virtual std::any sup() const = 0;
  virtual std::any inf() const = 0;

  virtual std::any plus(const std::any& v) const  = 0;
  virtual std::any minus(const std::any& v) const = 0;

  virtual std::any max(const std::any& l, const std::any& r) const  = 0;
  virtual std::any min(const std::any& l, const std::any& r) const  = 0;
  virtual std::any add(const std::any& l, const std::any& r) const  = 0;
  virtual std::any sub(const std::any& l, const std::any& r) const  = 0;
  virtual std::any mult(const std::any& l, const std::any& r) const = 0;
  virtual std::any div(const std::any& l, const std::any& r) const  = 0;
  virtual std::any mod(const std::any& l, const std::any& r) const  = 0;
  virtual std::any pow(const std::any& l, const std::any& r) const = 0;
};

template <typename T>
struct concrete_value_set : abstract_value_set
{
  ~concrete_value_set() override {}

  template <class U>
  std::any cast(std::any v) const /* override */
  {
    return {static_cast<U>(std::any_cast<T>(v))};
  }

  std::any max() const override { return {std::numeric_limits<T>::max()}; };
  std::any min() const override { return {std::numeric_limits<T>::min()}; };
  std::any sup() const override { return {std::numeric_limits<T>::max()}; };
  std::any inf() const override { return {std::numeric_limits<T>::min()}; };

  std::any plus(const std::any& v) const override { return {+std::any_cast<T>(v)}; };
  std::any minus(const std::any& v) const override { return {-std::any_cast<T>(v)}; };

  std::any max(const std::any& l, const std::any& r) const override
  {
    return {std::max(std::any_cast<T>(l), std::any_cast<T>(r))};
  }
  std::any min(const std::any& l, const std::any& r) const override
  {
    return {std::min(std::any_cast<T>(l), std::any_cast<T>(r))};
  }
  std::any add(const std::any& l, const std::any& r) const override
  {
    return {std::any_cast<T>(l) + std::any_cast<T>(r)};
  }
  std::any sub(const std::any& l, const std::any& r) const override
  {
    return {std::any_cast<T>(l) - std::any_cast<T>(r)};
  }
  std::any mult(const std::any& l, const std::any& r) const override
  {
    return {std::any_cast<T>(l) * std::any_cast<T>(r)};
  }
  std::any div(const std::any& l, const std::any& r) const override
  {
    return {std::any_cast<T>(l) / std::any_cast<T>(r)};
  }
  std::any mod(const std::any& l, const std::any& r) const override
  {
    return {std::any_cast<T>(l) % std::any_cast<T>(r)};
  }
  std::any pow(const std::any& l, const std::any& r) const override
  {
```

```
    return {std::pow(std::any_cast<T>(l), std::any_cast<T>(r))};
  }
};

struct type_erased_value
{
  type_erased_value(const std::any& v, const abstract_value_set& vs);

  const std::any&       val() const;
  const std::type_info& tid() const;

  template <typename From, typename To>
  type_erased_value cast(abstract_value_set* new_vs = nullptr) const
  {
    assert(typeid(From) == tid());
    auto new_val = std::any{static_cast<To>(std::any_cast<From>(v_))};
    return {new_val, new_vs != nullptr ? *new_vs : *vs_};
  }

  type_erased_value max() const;
  type_erased_value min() const;
  type_erased_value sup() const;
  type_erased_value inf() const;

  type_erased_value plus() const;
  type_erased_value minus() const;

  type_erased_value max(const type_erased_value& rhs) const;
  type_erased_value min(const type_erased_value& rhs) const;
  type_erased_value add(const type_erased_value& rhs) const;
  type_erased_value sub(const type_erased_value& rhs) const;
  type_erased_value mult(const type_erased_value& rhs) const;
  type_erased_value div(const type_erased_value& rhs) const;
  type_erased_value mod(const type_erased_value& rhs) const;
  type_erased_value pow(const type_erased_value& rhs) const;

private:
  std::any                  v_;
  const abstract_value_set* vs_;
};

template <>
struct value_set<type_erased_value> : abstract_value_set
{
  value_set(const type_erased_value& v);
  ~value_set() override;

  template <typename From, typename To>
  std::any cast(const std::any& v, abstract_value_set* new_vs = nullptr) const
  {
    auto absv = std::any_cast<type_erased_value>(v);
    return {absv.template cast<From, To>(new_vs)};
  }

  std::any max() const override;
  std::any min() const override;
  std::any sup() const override;
  std::any inf() const override;

  std::any plus(const std::any& v) const override;

  std::any minus(const std::any& v) const override;

  std::any max(const std::any& l, const std::any& r) const override;
  std::any min(const std::any& l, const std::any& r) const override;
  std::any add(const std::any& l, const std::any& r) const override;
  std::any sub(const std::any& l, const std::any& r) const override;
  std::any mult(const std::any& l, const std::any& r) const override;
  std::any div(const std::any& l, const std::any& r) const override;
  std::any mod(const std::any& l, const std::any& r) const override;
  std::any pow(const std::any& l, const std::any& r) const override;

private:
```

```
    const type_erased_value* v_;
  };
} // namespace mln
```

value__set.cpp

```
#include <mln/core/value/value_set.hpp>

#include <algorithm>
#include <any>
#include <cinttypes>
#include <cmath>
#include <limits>
#include <optional>
#include <type_traits>
#include <typeinfo>
#include <variant>


namespace details
{


  template <class... Args>
  auto any_to_variant_cast(std::any a) -> std::optional<std::variant<Args...>>
  {
    if (!a.has_value())
      return std::nullopt;

    std::optional<std::variant<Args...>> v = std::nullopt;

    bool found = ((a.type() == typeid(Args) && (v = std::any_cast<Args>(std::move(a)), true)) || ...);

    if (!found)
      return std::nullopt;

    return {std::move(*v)};
  }

  template <template <typename> class Op, typename... Args>
  auto call_template_op_from_type_info(const std::type_info& tid)
    -> std::optional<std::variant<Op<Args>...>>
  {
    std::optional<std::variant<Op<Args>...>> v = std::nullopt;

    bool found = ((tid == typeid(Args) && (v = Op<Args>{}, true)) || ...);

    if (!found)
      return std::nullopt;

    return {std::move(*v)};
  }

  template <class... Ts>
  struct type_list
  {
  };

  template <class... Ts>
  auto any_to_variant_cast_type_list(std::any a, type_list<Ts...>)
    -> std::optional<std::variant<Ts...>>
  {
    return any_to_variant_cast<Ts...>(a);
  }

  template <template <typename> class Op, typename... Ts>
  auto call_template_op_from_type_info_type_list(const std::type_info& tid, type_list<Ts...>)
      -> std::optional<std::variant<Op<Ts>...>>
  {
    return call_template_op_from_type_info<Op, Ts...>(tid);
  }

  using supported_types = type_list<std::uint8_t, std::uint16_t, std::uint32_t,
```

```
                                    std::uint64_t, std::int8_t, std::int16_t,
                                    std::int32_t, std::int64_t, float, double>;

  template <class T, class U, class C = std::common_type_t<T, U>>
  C max(const T& a, const U& b)
  {
    return std::max(static_cast<C>(a), static_cast<C>(b));
  }

  template <class T, class U, class C = std::common_type_t<T, U>>
  C min(const T& a, const U& b)
  {
    return std::min(static_cast<C>(a), static_cast<C>(b));
  }

  template <class T, class U>
  long double pow(const T& a, const U& b)
  {
    return std::pow(static_cast<long double>(a), static_cast<long double>(b));
  }

} // namespace details


namespace mln
{

  type_erased_value::type_erased_value(const std::any& v, const abstract_value_set& vs)
    : v_(v)
    , vs_(&vs)
  {
  }

  const std::any&       type_erased_value::val() const { return v_; }
  const std::type_info& type_erased_value::tid() const { return v_.type(); }

  template <typename T>
  struct limit_max
  {
    auto operator()() { return std::numeric_limits<T>::max(); }
  };

  type_erased_value type_erased_value::max() const
  {
    auto op = details::call_template_op_from_type_info_type_list<limit_max>(tid(),
                details::supported_types{});

    auto ret = std::visit([](auto op_) -> std::any { return {op_()}; }, *op);
    return {ret, *vs_};
  }

  template <typename T>
  struct limit_min
  {
    auto operator()() { return std::numeric_limits<T>::min(); }
  };

  type_erased_value type_erased_value::min() const
  {
    auto op = details::call_template_op_from_type_info_type_list<limit_min>(tid(),
                details::supported_types{});

    auto ret = std::visit([](auto op_) -> std::any { return {op_()}; }, *op);
    return {ret, *vs_};
  }

  type_erased_value type_erased_value::sup() const
  {
    auto op = details::call_template_op_from_type_info_type_list<limit_max>(tid(),
                details::supported_types{});

    auto ret = std::visit([](auto op_) -> std::any { return {op_()}; }, *op);
    return {ret, *vs_};
```

```
}

type_erased_value type_erased_value::inf() const
{
  auto op = details::call_template_op_from_type_info_type_list<limit_min>(tid(),
              details::supported_types{});

  auto ret = std::visit([](auto op_) -> std::any { return {op_()}; }, *op);
  return {ret, *vs_};
}

type_erased_value type_erased_value::plus() const { return {vs_->plus(v_), *vs_}; }
type_erased_value type_erased_value::minus() const { return {vs_->plus(v_), *vs_}; }

type_erased_value type_erased_value::max(const type_erased_value& rhs) const
{
  return {vs_->max(v_, rhs.val()), *vs_};
}
type_erased_value type_erased_value::min(const type_erased_value& rhs) const
{
  return {vs_->min(v_, rhs.val()), *vs_};
}
type_erased_value type_erased_value::add(const type_erased_value& rhs) const
{
  return {vs_->add(v_, rhs.val()), *vs_};
}
type_erased_value type_erased_value::sub(const type_erased_value& rhs) const
{
  return {vs_->sub(v_, rhs.val()), *vs_};
}
type_erased_value type_erased_value::mult(const type_erased_value& rhs) const
{
  return {vs_->mult(v_, rhs.val()), *vs_};
}
type_erased_value type_erased_value::div(const type_erased_value& rhs) const
{
  return {vs_->div(v_, rhs.val()), *vs_};
}
type_erased_value type_erased_value::mod(const type_erased_value& rhs) const
{
  return {vs_->mod(v_, rhs.val()), *vs_};
}
type_erased_value type_erased_value::pow(const type_erased_value& rhs) const
{
  return {vs_->pow(v_, rhs.val()), *vs_};
}


value_set<type_erased_value>::value_set(const type_erased_value& v)
  : v_(&v)
{
}

value_set<type_erased_value>::~value_set() {}

std::any value_set<type_erased_value>::max() const { return {v_->max()}; }
std::any value_set<type_erased_value>::min() const { return {v_->min()}; }
std::any value_set<type_erased_value>::sup() const { return {v_->sup()}; }
std::any value_set<type_erased_value>::inf() const { return {v_->inf()}; }

std::any value_set<type_erased_value>::plus(const std::any& v) const
{

  auto va = details::any_to_variant_cast_type_list(v, details::supported_types{});
  if (va)
    return {std::visit([](auto _v) -> std::any { return {+_v}; }, *va)};
  else
  {
    auto absv = std::any_cast<type_erased_value>(v);
    return {absv.plus()};
  }
}
```

```
std::any value_set<type_erased_value>::minus(const std::any& v) const
{
  auto va = details::any_to_variant_cast_type_list(v, details::supported_types{});
  if (va)
    return {std::visit([](auto _v) -> std::any { return {-_v}; }, *va)};
  else
  {
    auto absv = std::any_cast<type_erased_value>(v);
    return {absv.minus()};
  }
}

std::any value_set<type_erased_value>::max(const std::any& l, const std::any& r) const
{
  auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
  auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});

  if (lhs && rhs)
    return {
      std::visit([](auto lhs_, auto rhs_) -> std::any {
                   return {details::max(lhs_, rhs_)};
                 }, *lhs, *rhs)
    };
  else
  {
    auto absl = std::any_cast<type_erased_value>(l);
    auto absr = std::any_cast<type_erased_value>(r);
    return {absl.max(absr)};
  }
}

std::any value_set<type_erased_value>::min(const std::any& l, const std::any& r) const
{
  auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
  auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});

  if (lhs && rhs)
    return {
      std::visit([](auto lhs_, auto rhs_) -> std::any {
                   return {details::min(lhs_, rhs_)};
                 }, *lhs, *rhs)
    };
  else
  {
    auto absl = std::any_cast<type_erased_value>(l);
    auto absr = std::any_cast<type_erased_value>(r);
    return {absl.min(absr)};
  }
}

std::any value_set<type_erased_value>::add(const std::any& l, const std::any& r) const
{
  auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
  auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});

  if (lhs && rhs)
    return {
      std::visit([](auto lhs_, auto rhs_) -> std::any {
                   return {lhs_ + rhs_};
                 }, *lhs, *rhs)
    };
  else
  {
    auto absl = std::any_cast<type_erased_value>(l);
    auto absr = std::any_cast<type_erased_value>(r);
    return {absl.add(absr)};
  }
}

std::any value_set<type_erased_value>::sub(const std::any& l, const std::any& r) const
{
  auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
  auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});
```

```
  if (lhs && rhs)
    return {
      std::visit([](auto lhs_, auto rhs_) -> std::any {
                   return {lhs_ - rhs_};
                 }, *lhs, *rhs)
    };
  else
  {
    auto absl = std::any_cast<type_erased_value>(l);
    auto absr = std::any_cast<type_erased_value>(r);
    return {absl.sub(absr)};
  }
}

std::any value_set<type_erased_value>::mult(const std::any& l, const std::any& r) const
{
  auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
  auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});

  if (lhs && rhs)
    return {
      std::visit([](auto lhs_, auto rhs_) -> std::any {
                   return {lhs_ * rhs_};
                 }, *lhs, *rhs)
    };
  else
  {
    auto absl = std::any_cast<type_erased_value>(l);
    auto absr = std::any_cast<type_erased_value>(r);
    return {absl.mult(absr)};
  }
}

std::any value_set<type_erased_value>::div(const std::any& l, const std::any& r) const
{
  auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
  auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});

  if (lhs && rhs)
    return {
      std::visit([](auto lhs_, auto rhs_) -> std::any {
                   return {lhs_ / rhs_};
                 }, *lhs, *rhs)
    };
  else
  {
    auto absl = std::any_cast<type_erased_value>(l);
    auto absr = std::any_cast<type_erased_value>(r);
    return {absl.div(absr)};
  }
}

std::any value_set<type_erased_value>::mod(const std::any& l, const std::any& r) const
{
  auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
  auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});

  if (lhs && rhs)
    return {
      std::visit([](auto lhs_, auto rhs_) -> std::any {
                   return {lhs_ % rhs_};
                 }, *lhs, *rhs)
    };
  else
  {
    auto absl = std::any_cast<type_erased_value>(l);
    auto absr = std::any_cast<type_erased_value>(r);
    return {absl.mod(absr)};
  }
}

std::any value_set<type_erased_value>::pow(const std::any& l, const std::any& r) const
```

```
  {
    auto lhs = details::any_to_variant_cast_type_list(l, details::supported_types{});
    auto rhs = details::any_to_variant_cast_type_list(r, details::supported_types{});

    if (lhs && rhs)
      return {
        std::visit([](auto lhs_, auto rhs_) -> std::any {
                     return {details::pow(lhs_, rhs_)};
                   }, *lhs, *rhs)
      };
    else
    {
      auto absl = std::any_cast<type_erased_value>(l);
      auto absr = std::any_cast<type_erased_value>(r);
      return {absl.pow(absr)};
    }
  }

} // namespace mln
```

py__value__set.hpp

```
#include <mln/core/value/value_set.hpp>
#include <pybind11/pybind11.h>

#include <any>
#include <cinttypes>
#include <memory>
#include <string>
#include <string_view>
#include <typeinfo>
#include <variant>


namespace mln
{
  template <>
  struct value_set<pybind11::object> : abstract_value_set
  {
    value_set(pybind11::object python_vs_instance);

    ~value_set() override;

    template <typename U>
    std::any cast(const std::any& v) const;

    std::any max() const override;
    std::any min() const override;
    std::any sup() const override;
    std::any inf() const override;

    std::any plus(const std::any& v) const override;
    std::any minus(const std::any& v) const override;

    std::any max(const std::any& l, const std::any& r) const override;
    std::any min(const std::any& l, const std::any& r) const override;
    std::any add(const std::any& l, const std::any& r) const override;
    std::any sub(const std::any& l, const std::any& r) const override;
    std::any mult(const std::any& l, const std::any& r) const override;
    std::any div(const std::any& l, const std::any& r) const override;
    std::any mod(const std::any& l, const std::any& r) const override;
    std::any pow(const std::any& l, const std::any& r) const override;

  private:
    pybind11::object vs_instance_;
  };

} // namespace mln


namespace details
{
```

```
  template <class U>
  auto get_python_type()
  {
    // C++ type -> Python type
    static std::unordered_map<std::type_index, std::string> type_names{
        {std::type_index(typeid(bool{})), "bool"},          //
        {std::type_index(typeid(int8_t{})), "int"},         //
        {std::type_index(typeid(int16_t{})), "int"},        //
        {std::type_index(typeid(int32_t{})), "int"},        //
        {std::type_index(typeid(int64_t{})), "int"},        //
        {std::type_index(typeid(uint8_t{})), "int"},        //
        {std::type_index(typeid(uint16_t{})), "int"},       //
        {std::type_index(typeid(uint32_t{})), "int"},       //
        {std::type_index(typeid(uint64_t{})), "int"},       //
        {std::type_index(typeid(float{})), "float"},        //
        {std::type_index(typeid(double{})), "float"},       //
        {std::type_index(typeid((char*){})), "str"},        //
        {std::type_index(typeid((const char*){})), "str"}, //
        {std::type_index(typeid(std::string{})), "str"}};
    return type_names[std::type_index(typeid(U{}))];
  }

} // namespace details


namespace mln
{
  template <typename U>
  std::any value_set<pybind11::object>::cast(const std::any& v) const
  {
    pybind11::object v_ = std::any_cast<pybind11::object>(v);
    // if constexpr (std::is_arithmetic_v<U>) // bool, signed/unsigned int, floating point
    if constexpr (std::is_same_v<U, bool>) // bool
    {
      pybind11::bool_ pyv = vs_instance_.attr("cast")(v_, ::details::get_python_type<U>());
      return {static_cast<U>(pyv)};
    }
    else if constexpr (std::is_integral_v<U>) // signed/unsigned int
    {
      pybind11::int_ pyv = vs_instance_.attr("cast")(v_, ::details::get_python_type<U>());
      return {static_cast<U>(pyv)};
    }
    else if constexpr (std::is_floating_point_v<U>) // floating point
    {
      pybind11::float_ pyv = vs_instance_.attr("cast")(v_, ::details::get_python_type<U>());
      return {static_cast<U>(pyv)};
    }
    else
    {
      pybind11::object ret = vs_instance_.attr("cast")(v, ::details::get_python_type<U>());
      return {*(ret.cast<U*>())};
    }
  }

} // namespace mln
```

py_value_set.cpp

```
#include "py_value_set.hpp"

#include <any>

namespace mln
{

  value_set<pybind11::object>::value_set(pybind11::object python_vs_instance)
    : vs_instance_(python_vs_instance)
  {
  }

  value_set<pybind11::object>::~value_set()
  {
  }
```

```
std::any value_set<pybind11::object>::max() const { return {vs_instance_.attr("max")()}; }

std::any value_set<pybind11::object>::min() const { return {vs_instance_.attr("min")()}; }

std::any value_set<pybind11::object>::sup() const { return {vs_instance_.attr("sup")()}; }

std::any value_set<pybind11::object>::inf() const { return {vs_instance_.attr("inf")()}; }


std::any value_set<pybind11::object>::plus(const std::any& v) const
{
  auto pyv = std::any_cast<pybind11::object>(v);
  return {vs_instance_.attr("plus")(pyv)};
}

std::any value_set<pybind11::object>::minus(const std::any& v) const
{
  auto pyv = std::any_cast<pybind11::object>(v);
  return {vs_instance_.attr("minus")(pyv)};
}


std::any value_set<pybind11::object>::max(const std::any& l, const std::any& r) const
{
  auto pyl = std::any_cast<pybind11::object>(l);
  auto pyr = std::any_cast<pybind11::object>(r);
  return {vs_instance_.attr("max")(pyl, pyr)};
}

std::any value_set<pybind11::object>::min(const std::any& l, const std::any& r) const
{
  auto pyl = std::any_cast<pybind11::object>(l);
  auto pyr = std::any_cast<pybind11::object>(r);
  return {vs_instance_.attr("min")(pyl, pyr)};
}

std::any value_set<pybind11::object>::add(const std::any& l, const std::any& r) const
{
  auto pyl = std::any_cast<pybind11::object>(l);
  auto pyr = std::any_cast<pybind11::object>(r);
  return {vs_instance_.attr("add")(pyl, pyr)};
}

std::any value_set<pybind11::object>::sub(const std::any& l, const std::any& r) const
{
  auto pyl = std::any_cast<pybind11::object>(l);
  auto pyr = std::any_cast<pybind11::object>(r);
  return {vs_instance_.attr("sub")(pyl, pyr)};
}

std::any value_set<pybind11::object>::mult(const std::any& l, const std::any& r) const
{
  auto pyl = std::any_cast<pybind11::object>(l);
  auto pyr = std::any_cast<pybind11::object>(r);
  return {vs_instance_.attr("mult")(pyl, pyr)};
}

std::any value_set<pybind11::object>::div(const std::any& l, const std::any& r) const
{
  auto pyl = std::any_cast<pybind11::object>(l);
  auto pyr = std::any_cast<pybind11::object>(r);
  return {vs_instance_.attr("div")(pyl, pyr)};
}

std::any value_set<pybind11::object>::mod(const std::any& l, const std::any& r) const
{
  auto pyl = std::any_cast<pybind11::object>(l);
  auto pyr = std::any_cast<pybind11::object>(r);
  return {vs_instance_.attr("mod")(pyl, pyr)};
}

std::any value_set<pybind11::object>::pow(const std::any& l, const std::any& r) const
```

```
  {
    auto pyl = std::any_cast<pybind11::object>(l);
    auto pyr = std::any_cast<pybind11::object>(r);
    return {vs_instance_.attr("pow")(pyl, pyr)};
  }

} // namespace mln
```

D.3.3 Mathematical morphology routines

morpho.hpp

```
#include <pybind11/pybind11.h>


namespace mln::py
{

  void init_module_morpho(pybind11::module& m);

}
```

morpho.cpp

```
#include "morpho.hpp"

#include "ndimage.hpp"
#include "py_value_set.hpp"
#include "visit.hpp"

#include <mln/core/algorithm/transform.hpp>
#include <mln/core/image/ndimage.hpp>
#include <mln/core/se/disc.hpp>
#include <mln/core/se/rect2d.hpp>
#include <mln/morpho/experimental/dilation.hpp>

#include <fmt/core.h>

#include <pybind11/stl.h>

#include <algorithm>
#include <string_view>
#include <variant>


namespace my
{
  template <class T>
  mln::image2d<float> slow_stretch(const mln::image2d<T>& src,
                                   const mln::value_set<pybind11::object>& py_vs)
  {
    mln::image2d<float> res = mln::transform(src, [&py_vs](auto val) -> float {
      auto anymax  = py_vs.max();                    // returns std::any of pybind11:object
      auto anyval  = std::any{pybind11::cast(val)};  // converts to std::any of pybind11::object
      auto anyret  = py_vs.div(anyval, anymax);      // returns std::any of pybind11::object
      auto anyfret = py_vs.template cast<float>(anyret); // returns std::any of float
      return std::any_cast<float>(anyfret);          // returns float
    });
    return res;
  }

  template <class T>
  mln::image2d<float> virtual_dispatch_stretch(const mln::image2d<T>& src)
  {
    auto                vs   = mln::concrete_value_set<T>{};     // value-set for T
    auto                vs_f = mln::concrete_value_set<float>{}; // value-set for float
    mln::image2d<float> res  = mln::transform(src, [&vs, &vs_f](auto val) -> float {
      auto anymax  = vs.max();                              // returns std::any
      auto fanyval = vs.template cast<float>(val);          // cast to float in std::any
      auto fanymax = vs.template cast<float>(anymax);       // cast to float in std::any
      return std::any_cast<float>(vs_f.div(fanyval, fanymax)); // div returns float
```

```
    });
    return res;
  }

  template <class T>
  mln::image2d<float> stretch_virtual_dispatch_type_erased_value(const mln::image2d<T>& src)
  {
    auto                vs   = mln::concrete_value_set<T>{};     // value-set for T
    auto                vs_f = mln::concrete_value_set<float>{}; // value-set for float
    mln::image2d<float> res  = mln::transform(src, [&vs, &vs_f](auto val) -> float {
      // simulate having an image<type_erased_value>
      auto anyval = std::any{val}; // std::any of T
                                   // type_erased_value of std::any of T aware of value-set of T
      auto abs_anyval = mln::type_erased_value{anyval, vs};
      // instantiate a value-set for type_erased_value
      auto abs_vs = mln::value_set<mln::type_erased_value>{abs_anyval};
      auto anyabs_anymax =
          abs_vs.max(); // returns std::any of type_erased_value
                        // cast underlying std::any of type_erased_value of std::any of T into
                        // std::any of type_erased_value of std::any of float
                        // aware of value-set for float
      auto anyabs_fanyval = abs_vs.template cast<T, float>(std::any{abs_anyval}, &vs_f);
      auto anyabs_fanymax = abs_vs.template cast<T, float>(anyabs_anymax, &vs_f);
      // dispatch on known type, find a type_erased_value, then call
      // anyabs_fanyval.div(anyabs_fanymax) to perform division which will call
      // the underlying value-set for float for this operation
      auto anyabs_fanyret = abs_vs.div(anyabs_fanyval, anyabs_fanymax);
      // convert result back into float for returning to the image
      auto anyfret = std::any_cast<mln::type_erased_value>(anyabs_fanyret).val();
      return std::any_cast<float>(anyfret);
    });
    return res;
  }

  template <class T>
  mln::image2d<float> fast_stretch(const mln::image2d<T>& src)
  {
    auto                vs   = src.get_value_set();     // value-set for T
    auto                vs_f = mln::value_set<float>{}; // fast value-set for float
    mln::image2d<float> res  = mln::transform(src, [&vs, &vs_f](auto val) -> float {
      auto max  = vs.max();                    // returns T
      auto fval = vs.template cast<float>(val); // returns float
      auto fmax = vs.template cast<float>(max); // returns float
      return vs_f.div(fval, fmax);              // div directly returns float
    });
    return res;
  }
} // namespace my

namespace
{

  template <typename T>
  struct stretch_operator_t
  {
    template <typename Img>
    mln::ndbuffer_image operator()(Img&& img, const std::optional<pybind11::object>& py_vs) const
    {
      if (auto* image_ptr = std::forward<Img>(img).template cast_to<T, 2>(); image_ptr)
      {
        if (py_vs.has_value())
        {
          auto vs = mln::value_set<pybind11::object>{py_vs.value()};
          return my::slow_stretch(*image_ptr, vs);
        }
        else
        {
          return my::fast_stretch(*image_ptr);
        }
      }
      else
      {
        std::runtime_error("Unable to convert the image to the required type.");
```

```
      return {};
    }
  }
};

mln::ndbuffer_image stretch(mln::ndbuffer_image input, std::optional<pybind11::object> py_vs)
{
  if (input.pdim() == 2)
  {
    return mln::py::visit<stretch_operator_t>(input.sample_type(), input, py_vs);
  }
  else
  {
    std::runtime_error("Unsupported dimension.");
    return {};
  }
}

template <typename T>
struct stretch_virtual_dispatch_operator_t
{
  template <typename Img>
  mln::ndbuffer_image operator()(Img&& img) const
  {
    if (auto* image_ptr = std::forward<Img>(img).template cast_to<T, 2>(); image_ptr)
    {
      return my::virtual_dispatch_stretch(*image_ptr);
    }
    else
    {
      std::runtime_error("Unable to convert the image to the required type.");
      return {};
    }
  }
};

mln::ndbuffer_image stretch_virtual_dispatch(mln::ndbuffer_image input)
{
  if (input.pdim() == 2)
  {
    return mln::py::visit<stretch_virtual_dispatch_operator_t>(input.sample_type(), input);
  }
  else
  {
    std::runtime_error("Unsupported dimension.");
    return {};
  }
}

template <typename T>
struct stretch_virtual_dispatch_type_erased_value_operator_t
{
  template <typename Img>
  mln::ndbuffer_image operator()(Img&& img) const
  {
    if (auto* image_ptr = std::forward<Img>(img).template cast_to<T, 2>(); image_ptr)
    {
      return my::stretch_virtual_dispatch_type_erased_value(*image_ptr);
    }
    else
    {
      std::runtime_error("Unable to convert the image to the required type.");
      return {};
    }
  }
};

mln::ndbuffer_image stretch_virtual_dispatch_type_erased_value(mln::ndbuffer_image input)
{
  if (input.pdim() == 2)
  {
    return mln::py::visit<stretch_virtual_dispatch_type_erased_value_operator_t>(
            input.sample_type(), input);
```

```
    }
    else
    {
      std::runtime_error("Unsupported dimension.");
      return {};
    }
  }

  using se_t = std::variant<mln::se::disc, mln::se::disc_non_decomp,
                            mln::se::rect2d, mln::se::rect2d_non_decomp>;

  template <typename T>
  struct dilate2d_operator_t
  {
    template <typename Img, typename SE>
    mln::ndbuffer_image operator()(Img&& img, SE&& se) const
    {
      if (auto* image_ptr = std::forward<Img>(img).template cast_to<T, 2>(); image_ptr)
      {
        return mln::morpho::experimental::dilation(*image_ptr, std::forward<SE>(se));
      }
      else
      {
        std::runtime_error("Unable to convert the image to the required type.");
        return {};
      }
    }
  };

  mln::ndbuffer_image dilate(mln::ndbuffer_image input, const se_t& se)
  {
    if (input.pdim() == 2)
    {
      return std::visit(
          [&input](const auto& se_) {
            return mln::py::visit<dilate2d_operator_t>(input.sample_type(), input, se_);
          }, se);
    }
    else
    {
      std::runtime_error("Unsupported dimension.");
      return {};
    }
  }

} // namespace


namespace mln::py
{

  using namespace pybind11::literals;

  void init_module_morpho(pybind11::module& m)
  {
    m.def("dilate", dilate,
          "Perform a morphological dilation.\n"
          "\n"
          "structuring element must be valid.",
          "Input"_a, "se"_a)

      .def("stretch", stretch,
           "Perform a morphological stretch.\n"
           "Input"_a,
           "vs"_a = pybind11::none())

      .def("stretch_virtual_dispatch", stretch_virtual_dispatch,
           "Perform a morphological stretch.\n"
           "Input"_a)

      .def("stretch_virtual_dispatch_type_erased_value", stretch_virtual_dispatch_type_erased_value,
           "Perform a morphological stretch.\n"
           "Input"_a);
```

```
  }

} // namespace mln::py
```

D.4 Exposed Pylena main module

pylena.cpp

```cpp
#include "morpho.hpp"
#include "ndimage.hpp"
#include "se.hpp"

#include <pybind11/pybind11.h>


PYBIND11_MODULE(pylena, m)
{
  mln::py::init_pylena_numpy(m);

  auto mmorpho = m.def_submodule("morpho", "Mathematical morphology module.");
  mln::py::init_module_morpho(mmorpho);

  auto mse = m.def_submodule("se", "Structuring elements module.");
  mln::py::init_module_se(mse);
}
```

D.5 Python value set module

The module has the following directory structure:

```
ValueSetExample/
| __init__.py
| BaseValueSet.py
| ExampleValueSet.py
```

BaseValueSet.py

```
from abc import ABC, abstractmethod
from typing import Any


class AbstractValueSet(ABC):

    def __init__(self): pass

    @abstractmethod
    def cast(self, value: Any, type_):
        if type_ in ["int", "float", "bool", "str"]:
            import importlib
            module = importlib.import_module('builtins')
            cls = getattr(module, type_)
            return cls(value)
        raise ValueError()

    @abstractmethod
    def max(self):
        import math
        return math.inf

    @abstractmethod
    def min(self):
        import math
        return -math.inf

    @abstractmethod
    def sup(self):
        import math
        return math.inf

    @abstractmethod
    def inf(self):
        import math
        return -math.inf

    @abstractmethod
    def plus(self, value: Any) -> Any:
        return + value

    @abstractmethod
    def minus(self, value: Any) -> Any:
        return - value

    @abstractmethod
    def add(self, lhs: Any, rhs: Any) -> Any:
        return lhs + rhs

    @abstractmethod
    def sub(self, lhs: Any, rhs: Any) -> Any:
        return lhs - rhs

    @abstractmethod
    def mult(self, lhs: Any, rhs: Any) -> Any:
        return lhs * rhs

    @abstractmethod
    def div(self, lhs: Any, rhs: Any) -> Any:
        return lhs / rhs
```

```
    @abstractmethod
    def mod(self, lhs: Any, rhs: Any) -> Any:
        return lhs % rhs

    @abstractmethod
    def pow(self, lhs: Any, rhs: Any) -> Any:
        return lhs ** rhs
```

ExampleStruct.py

```
from typing import Any


class MyStruct:
    v_: Any

    def __init__(self, v: Any):
        self.v_ = v

    def getV(self) -> Any:
        return self.v_

    def setV(self, v: Any):
        self.v_ = v
```

ExampleValueSet.py

```
from ValueSetExample.ExampleStruct import MyStruct
from ValueSetExample.BaseValueSet import AbstractValueSet

from typing import Any


class MyValueSet(AbstractValueSet):

    def __init__(self): pass

    def get_MyStruct__(self, v: Any):
        return v.getV() if isinstance(v, MyStruct) else v

    def cast(self, value: Any, type_):
        v = self.get_MyStruct__(value)
        return super().cast(v, type_)

    def max(self):
        return 255

    def min(self):
        return 0

    def sup(self):
        return 255

    def inf(self):
        return 0

    def plus(self, value: Any) -> Any:
        v = self.get_MyStruct__(value)
        return MyStruct(super().plus(v))

    def minus(self, value: Any) -> Any:
        v = self.get_MyStruct__(value)
        return MyStruct(super().minus(v))

    def add(self, lhs: Any, rhs: Any) -> Any:
        l = self.get_MyStruct__(lhs)
        r = self.get_MyStruct__(rhs)
        return MyStruct(super().add(l, r))

    def sub(self, lhs: Any, rhs: Any) -> Any:
        l = self.get_MyStruct__(lhs)
        r = self.get_MyStruct__(rhs)
```

```
        return MyStruct(super().sub(l, r))

    def mult(self, lhs: Any, rhs: Any) -> Any:
        l = self.get_MyStruct__(lhs)
        r = self.get_MyStruct__(rhs)
        return MyStruct(super().mult(l, r))

    def div(self, lhs: Any, rhs: Any) -> Any:
        l = self.get_MyStruct__(lhs)
        r = self.get_MyStruct__(rhs)
        return MyStruct(super().div(l, r))

    def mod(self, lhs: Any, rhs: Any) -> Any:
        l = self.get_MyStruct__(lhs)
        r = self.get_MyStruct__(rhs)
        return MyStruct(super().mod(l, r))

    def pow(self, lhs: Any, rhs: Any) -> Any:
        l = self.get_MyStruct__(lhs)
        r = self.get_MyStruct__(rhs)
        return MyStruct(super().pow(l, r))
```

___init___.py

```
from ValueSetExample.BaseValueSet import AbstractValueSet
from ValueSetExample.ExampleStruct import MyStruct
from ValueSetExample.ExampleValueSet import MyValueSet
```

D.6 Benchmark source code

benchmark.py

```
import numpy as np
import skimage
import timeit
import pylena as pln
from ValueSetExample import MyValueSet, MyStruct

grayscale = skimage.data.camera()

vs = MyValueSet()


def test_fast():
    global grayscale
    pln_ret = pln.morpho.stretch(grayscale)
    return pln_ret


def test_slow():
    global grayscale, vs
    pln_ret = pln.morpho.stretch(grayscale, vs)
    return pln_ret


def test_virtual_dispatch():
    global grayscale
    pln_ret = pln.morpho.stretch_virtual_dispatch(grayscale)
    return pln_ret


def test_virtual_dispatch_type_erased_value():
    global grayscale
    pln_ret = pln.morpho.stretch_virtual_dispatch_type_erased_value(grayscale)
    return pln_ret


if __name__ == "__main__":
    print("Benchmarking static-dynamic bridge...")
    cpp_fast_ret = timeit.timeit("test_fast()", globals=locals(), number=10)
    cpp_virtual_dispatch_ret = timeit.timeit(
        "test_virtual_dispatch()", globals=locals(), number=10)
    cpp_virtual_dispatch_type_erased_value_ret = timeit.timeit(
        "test_virtual_dispatch_type_erased_value()", globals=locals(), number=10)
    cpp_python_vs_ret = timeit.timeit(
        "test_slow()", globals=locals(), number=10)
    print("Benchmarking done...")

    print("Native value-set with native C++ value-type: {0:.4f}sec"
          .format(cpp_fast_ret))
    print("Value-set with virtual dispatch with native C++ value-type: {0:.4f}sec"
          .format(cpp_virtual_dispatch_ret))
    print("Value-set with virtual dispatch with C++ type-erased values: {0:.4f}sec"
          .format(cpp_virtual_dispatch_type_erased_value_ret))
    print("Injected Python value-set with native C++ value-type: {0:.4f}sec"
          .format(cpp_python_vs_ret))
```