Advanced C++ Programming Course Project Report

Python bindings for Pylene

Aymeric Fages

Nicolas BLIN

Pierrick Made

Théo LEPAGE

Youssef Benkirane

July 2020

Contents

1 Introduction

This report will compile what has been done for the CPPA project, by the group composed of Aymeric Fages, Nicolas BLIN, Pierrick Made, Théo LEPAGE and Youssef Benkirane. The project started on June 29th, 2020 and was due on July 19th, 2020.

The aim of this project is to create Python bindings for Pylene, a C++ image processing library. Throughout the project, the main problem is to properly handle static and dynamic polymorphism.

This report will present the architecture of our project, explain how the Python bindings were implemented and provide a solution to the problem stated above.

2 Project structure

2.1 The compilation step

All necessary steps to build the project are described in the README. We rely on CMake and Conan which is a dependency and package manager for C++. As our two main dependencies are Pylene and Pybind, they are included in the file `conanfile.txt` at the root of the project. The file `CMakeLists.txt` contains the build configuration including the compilation flags and the language standard used as we want to enable all warnings and use C++ 20 standard. Our library, which is a shared object for Python, is generated by a target created with `pybind11_add_module(...)`. It is noteworthy that to link pylene to the generated library we have to compile it with the `-fPIC` compilation flag in order to enable "Position Independent Code" which means that the generated is not dependent on being located at a specific address. This issue was fixed by adding the option `pylene:shared=True` to the conanfile.

2.2 Overview of the project architecture

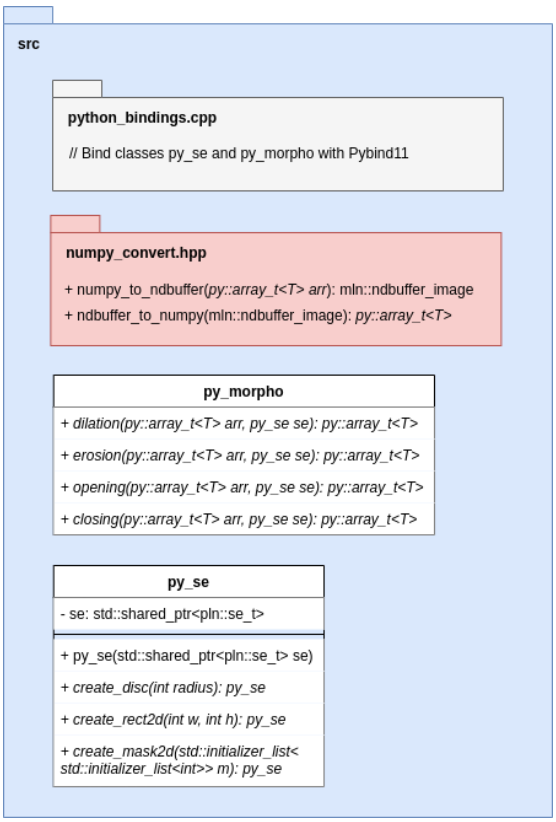

Figure 1: UML diagram of source files in src/

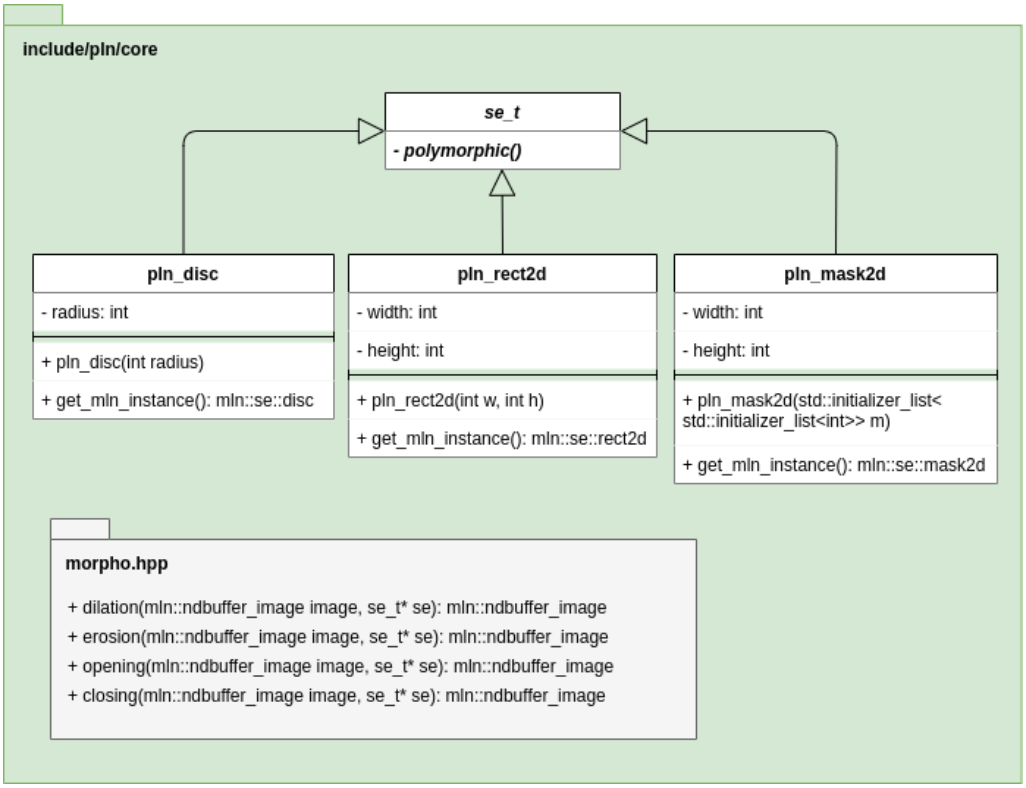

Figure 2: UML diagram of source files in include/mln/core/

As shown on Figure 1 and 2, the project is divided into two parts: the source files, located in the folder `src`, used for the python bindings and the conversion between python arrays and data structures used by Pylene ; the classes and methods added to the Pylene library.

src The entry point of the project is the file `python_bindings.cpp` which binds classes py_se and py_morpho used to provide methods directly accessible through Python. The details regarding this implementation will be described in the next section.

include/pln/core In order to keep the project well structured, features that should belong to the Pylene library are located in a separate folder. The different calls to the functions of Pylene are present in the file `morpho.hpp`. The UML diagram in Figure 2 gives a good representation of the hierarchical structure between classes `pln_disc`, `pln_rect2d`, `pln_mask2d` and `se_t` described later. More details about the type erasure will be given in section three.

2.3 Our test suite

We created another CMake target so that the command `make test` build the library and launch our functional tests. The generated library is first copied in the tests folder and then our python test suite, based on unittest, is launched using `python -m unittest`. The main goal of our tests is to compare the generated images between our python bindings and the usual way to use Pylene. The tests folder contains sample images generated after a dilation, an erosion, an opening, a closing and with various parameters for the structuring element used. Our tests generate generate images with our own library and compare them to the reference using mean-squared error (MSE). There is also a debug mode available to show images during the execution, it can be activated by toggling the variable `ALWAYS_SHOW_IMAGES` in the python code.

3 Creating Python bindings

3.1 py_morpho and py_se

In order to create a convenient user experience and to fit with the subject requirements, classes `py_morpho` and `py_se` are used to provide Pylene features through Python.

Instantiating a structuring element From the user side, static methods `pln.se.disc(my_radius)` and `pln.se.rectangle(width=my_width, height=my_height)` will first create the specified instance (`pln::pln_disc` or `pln::pln_rect2d`). Then, they return a `py_se` which holds a shared pointer on this newly created instance as a `pln::se_t`.

Applying a morphology operation on a numpy array Again from the user side, static methods like `pln.morpho.dilation(image, py_se)` will be responsible of converting the numpy array back and forth and call the specified method: `pln::dilation`, `pln::erosion`, `pln::opening` and `pln::closing`. At this step, the structuring element given to these methods is the pointer to a `pln::se_t` which was stored in the `py_se`.

3.2 Converting numpy arrays to `mln::ndbuffer_image`

To convert a `py::array_t` to a `mln::ndbuffer_image` we simply call the function `mln::ndbuffer_image::from_buffer` with the following arguments:

- **buffer**: The pointer to the data of the numpy array casted with a `static_cast<std::byte*>`.
- **sample_type**: A `mln::sample_type_id` representing the type of the values in the buffer. We retrieve this value depending on the templated type T of the function using `mln::sample_type_traits<T>::id()`.
- **dim**: The number of dimensions of the input array.
- **sizes**: The shape of the input array converted from a `std::vector<ssize_t>` a `int[]`.
- **strides**: `nullptr` in order to let the function `from_buffer` automatically deduce the strides.
- **copy**: `true` even if, according to us, the buffer is already a copy but using `false` was causing segmentation faults.

For the reverse conversion, we simply do a similar treatment by filling a `py::buffer_info` structure to construct the `py::array_t`. We also make sure to reverse the order of the strides vector as it is not represented the same way by Pylene.

To handle RGB images, we consider that a numpy array of dimension 3 with 3 elements in the last dimension is a 2D `mln::ndbuffer_image` with a sample type of `mln::sample_type_id::`RGB8. In other situations with incompatible dimensions, an exception is thrown.

3.3 The problem of initializing a mask2d

One issue arised when trying to handle the structuring element mask2D : it can only be built using an initalizer_list of initalizer_list. Initializer_list are inherently static, to declare them you need to write them as such in the code (). You cannot dynamically create them from a vector or via a push or insert method. To cope with the issue, we tried a solution mixing variadic templates and static declaration but didn't succeed. We found online a piece of code (that can be found in vector_to_init_list.hpp) that converted a vector to an initalizer list. We tried to use this function to handle a fixed range of list initializer using a switch and only handle size from 1 to 5. Unfortunately, we are not able to use their value as their lifetime is limited like many other temporary objects.

3.4 Factorizing binding multiple types

To avoid having 4×7 lines, 4 for the 4 functions (dilation, erosion, opening, closing), 7 for the 7 types (uint8/16/32/64, bool, double/float), we use variadic template. The goal is to have a single function binding our 4 functions to all the types passed by template. To do se we use list-initlizer expansion. We build a dummy array using the list-initlizer syntax. In each cell of the array we using the (statement, expression) syntax. This allows us to first, call our single template function binding for one type our 4 functions. It then return 0, this allows us to store a valid value in the array. The unpacking (using ... syntaxe) builds a sequence of (statement, expression) which are all valid in our array. This allows ultimately to call our binding function on all of our template parameter.

We are able (due to types problems) to also generalize this to the functions, i.e store into an array the functions that need binding.

4 Pylene dynamic wrappers and the type erasure

4.1 Type erasure

4.1.1 The need for type erasure

```
namespace pln::morpho {
    mln::ndbuffer_image dilation(mln::ndbuffer_image input, se_t se);
}
```

Figure 3: Signature for the dilation function

As shown on figure 3, the four functions that we will need (dilation, erosion, opening and closing) will take as argument an image and an object of type set_t which stands for *structuring element type*. A structuring element can be of any shape: a disk, a rectangle, a mask or any other shape which have their own types. However, this function requires that we abstract all these type into a single one, a set_t type, which true type will be deduce at run-time. To do so, we need to use type erasure.

4.1.2 A solution for C++ type erasure

There is several ways to do type erasure in C++. The first thing we did was to check what all structuring elements have in common. According to pylene, the four functions that we will use can take any object as a structuring element argument as long as this latter satisfies the constraints of one of the pylene concept named "StructuringElement". Would the set_t type then just be an alias on a StructuringElement object ? Even though this would greatly make our lives easier, it not, since the Structuring Element concept from the pylene library is templated, meaning that it uses types resolved at compile time. So here we are, trying to define an argument type for our functions using statically resolved typenames in a very dynamic context, which is impossible. Note that the fact that we are unable to describe a generic type for our structuring elements as given to pylene functions will create even more issues as described in section 4.2.

Even though we cannot get a generic type for structuring elements to give to pylene functions, we still would like to have a generic set_t type. A naive C oriented solution for that would be to create a structure containing an union of each possible classes the set_t class can be, and an enumeration of those types. Another C-like approach would be to store a *void** pointer on one of the possible class that the set_t class can be combined with an enumeration saying which type it actually is. Since we are doing modern C++, we want to think of a more clever way.

A good way to do type erasure is to use inheritance: we can define the set_t as a parent class, which children are the classes describing each structuring element, i.e. a rectangle class, a disk class... This

method is actually close to the one using unions except that it uses C++ semantic of inheritance. So we created three class to represent our three structuring elements, and the set_t class that is the parent class of all the three others. We then use *dynamic casts* in order to resolve the actual type of a set_t class.

4.2 Call to the pylene library

Writing our 4 pln functions (dilation, erosion, opening, closing) indubitably inferred writing a lot of code duplication.

Indeed, inside each morphological function we needed a first set of if statements to select the correct image type (double, rgb...) and then inside each function another set of if statements to select the correct structuring element. To solve the former we pass the type by template parameter. This is possible because python can map the type correctly using pybind (np.uint8 to uint8 for example). The only type we cannot handle is rgb8 since python does not handle it by default. This forces us to duplicate at least one $if/else$ statement in the 4 functions. We could not do it before, since our 4 functions need to take an ndbuffer. Delegating it to the calling function would still result in a $if/else$ statement in the 4 functions.

Inside each function the is also 3 other if statements to handle the 3 structuring-element types. Again, we cannot put it beforehand because of the same two reasons. Our function need to take an se_t type (and not the concrete type) and this would still result in 3 if in 4 function in the calling code. One idea was to first cast it in a separate function and call this function in our 4 functions. The issue is, how to type such a function. The return type cannot be se_t because the goal is to get rid of it, it cannot be one of our 3 types because the function can return one of the 3 and it cannot be the concept "Structure Element" because it is itself templated with the concrete type. One possible other way to solve this problem was to use function pointer. Delegate the conversion and execution of the function to one function that would be called by our 4 morphological functions. Again, type problem prevents us from doing so. Indeed, those 4 function are templated. Casting them inside a std::function would mean choosing the structuring element template type (as it is part of the 4 functions signature). Nonetheless, at compile time, we do not yet know the type of the structuring element since the conversion is being done dynamically.

The issue here is linked to the danger of code duplication. With 10 functions and 10 structuring element types, adding one new type would result in 10 new lines in 10 functions with the risk of forgetting one. Since we can't factorize using a function, the best way to avoid the dangers of code duplication is to let the pre-processor do it itself using define statements. This is clearly not pure beautiful modern C++ but it has the benefit of avoiding the code duplication problems.

5 Conclusion

Although our Python library provide most of the bindings required by the subject, the features below were not developed in time or were not considered.

1. Instantiating a mask from Python as it implies creating a `std::initializer_list` from a `std::vector`.
2. Handling numpy arrays with a dimension different from 2 or 3. Concerning 3D numpy arrays, only those with 3 elements on the last dimension are supported to allow only RGB images as input.

We made our best so that our final library makes applying mathematical morphology operations on numpy arrays convenient through Python. Furthermore, it passes all of our functional tests on color and grayscale images which implies that it produces expected results.