National Institute of Applied Sciences and Technology

CARTHAGE UNIVERSITY

Graduation Project

Specialty : **Software Engineering**

Aare: a flexible data analysis library for hybrid pixel detectors

Presented by

Bechir Braham

INSAT Supervisor : **Dr. Guesmi Ghada**
Company Supervisor : **Dr. Fröjd Erik**

Presented on : **??/??/2024**

JURY

M. President FLEN (President)
Ms. Reviewer FLENA (Reviewer)

Academic Year : 2023/2024

Acknowledgements

Table of Contents

List of Figures

List of Tables

Abstract

This is the english abstract of your project. It must be longer and presented in more details than the abstract you write on the back of your report.

General Introduction

–TODO–

Part I

Chapter 1

Chapter I

Project Context and Scope

Summary

Introduction

In this first chapter, we introduce the Paul Scherrer institut and its detectors group. We explain the need for our project and the problems it aims to solve. Furthermore, we will go through the goals of the project and provide a high-level timeline capturing its progression.

1 Presentation of The Host Company

The Paul Scherrer Institute (PSI) is the largest research institute for natural and engineering sciences within Switzerland. Created in 1998, the institute is located in Canton Aargau and employs around 2300 people. PSI is composed of 8 main research centers:

- Center for Life Sciences
- Center for Neutron and Muon Sciences
- Center for Nuclear Engineering and Sciences
- Center for Energy and Environmental Sciences
- Center for Photon Science
- Center for Scientific Computing
- Theory and Data
- Center for Accelerator Science and Engineering.

In addition the Paul Scherrer Institute is famous for its synchotron: the Swiss Light Source (SLS). A synchotron is a charged particle accelerator that accelerates electrons to nearly the speed of light. The electrons are then forced to travel in a circular path, and emit synchrotron radiation in the form of x-rays. Scientists use the x-rays to study the properties of materials, and to perform experiments in a wide range of fields.
The SLS is a third-generation synchrotron light source, which provides high-brilliance photon beams with high spectral resolution and tunable energy. It is used for reseach in materials science, biology, chemistry and more. [Bög02; Web; Lv2222].

2 Detector's Group Presentation and Work

The Detector's Group is one of the research groups at the Paul Scherrer Institute. It is part of the Laboratory for X-ray Nanoscience and Technologies (LXN) which itself is part of the Center for Photon Science.
The group is responsible for the development of new detectors, the maintenance of the existing ones, and the development of software for the data acquisition and analysis. The group is also involved in the development of new techniques for the data analysis, such as image processing, pattern recognition, and machine learning. These Detector's are an integral part of the beamline's setup, and are used to detect the x-ray photons that are emitted by the synchrotron. Many experiments in different fields can use these detectors such as for studying the properties of materials [But+24], protein structures [Pom+09], crystallography [Leo+23], biology [Lem+23; Dul+24], and many other fields of research.
On the software side, the group is responsible for the development of the software that is used to control the detectors, acquire the data, and analyze it. The software is developed in C++

Figure I.1 – A picture of the Paul Scherrer Institute. The SLS is the large circular building. PSI has two main parts the East and West separated by the Aare river.

and is used by scientists to perform their experiments. The main package maintained by the detector's group is the "SLS Detector Package" available publicly on https://github.com/slsdetectorgroup/slsDetectorPackage. The package provides several binaries such as:

- **slsReceiver** The receiver server acquires incoming data from detectors using UDP and listens for configurations from host machine using TCP.
- **slsDetectorGet** Used by the host machine to request the configuration on the Receiver or the Detector.
- **slsDetectorPut** Used to configure parameters on both the Receiver and Detector.
- **slsDetectorGui** A graphical user interface (GUI) that receives data from the Receiver and displays it.

The list of binaries is not exhaustive, and the package contains many other binaries for simulating detectors, analyzing data, and many other utilities.

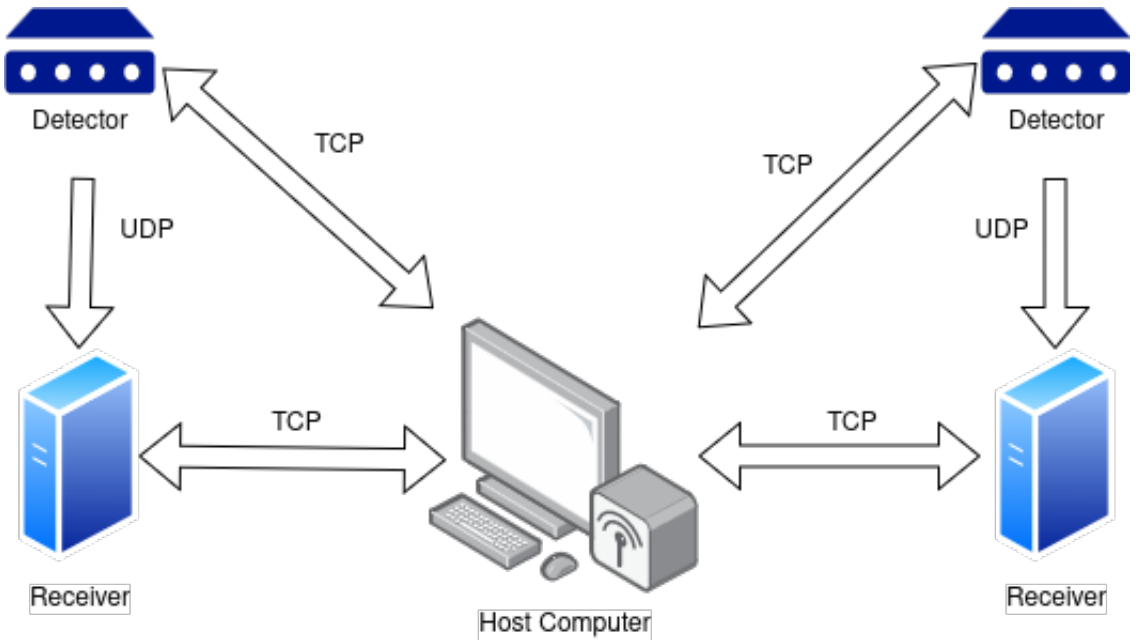


Figure I.2 – slsDetectorPackage setup for two detectors. Configuration uses TCP while data streaming from detector to slsReceiver uses UDP

3 Problem Statement

3.1 Existing Solution

The detector's group includes scientists, software engineers, firmware engineers, chip designers and many more roles. The group is diverse and the libraries' usage differs from one user to another. In general the usage of the libraries includes: acquiring data from network , configuring receivers and detectors, storing incoming data, processing data on the fly or after storing it. For the standard functionalities users rely on the slsDetectorPackage binaries. But the slsDetectorPackage is a generic software developed for public use and has very broad functionalities. Hence, for specific use cases scientists might need to write their own scripts or change the slsDetectorPackage source code and build again.

3.2 Limits of The Existing Solution

The slsDetectorPackage is a very powerful software package, but it has some major limitations.

3.2.1 Code Complexity

First, the code is very complex and has a steep learning curve. The code is written in C++ and uses many advanced features of the language. The code is also very large, with around 200 thousand lines of code. This makes it difficult for new users to understand how the code works, and to modify it to suit their needs. In addition, scientists are not software engineers, and in case of a bug, new feature or a specific use case they will be exposed to complex code that they are not familiar with. This might include the need to understand C++ code, multi-threading, network programming, and many other advanced topics.

3.2.2 Code Rigidity

Second, the code is very rigid and inflexible. The code is designed to work in a specific way, and it is difficult to modify it to work in a different way. This means that scientists are limited in what they can do with the code, and they are forced to work within the constraints of the existing code. This can be very frustrating for scientists, who may have specific requirements that are not met by the existing code. Furthermore, some of the speicific use case implementations are very brittle and can break easily if the code is modified. It lacks proper testing, error handling and logging. This makes it difficult to maintain and extend the code.

3.2.3 Code Duplication

Third, the code is duplicated in many places. Scientist often rely on their own scripts to perform specific tasks. This leads to each scientist having their own version of the code, which is difficult to maintain and update. and also results in the use of sub-optimal code, which is not efficient or reliable.

3.2.4 Data Storage Limitations

As the detectors become more and more advanced, the amount of data that they produce is increasing. This has made processing the incoming data in real-time a must. The slsDetectorPackage provides limited functionalities for processing data on the fly. This means that scientists need to store the data on disk and process it later. This is not ideal and is becoming less practical as the amount of data can be very expensive to store and process. The new library should be designed to process around 10GB/s of data in real-time.

4 Project Goals

The goal of this project is to develop a new library that will address the limitations of the existing software. The new software package will be designed to be simple, flexible, and efficient. It will be easy to use, and will be designed to meet the needs of scientists and engineers.

The library should include functionalities for acquiring data from receiver servers, stream data to receivers, read and write raw data files and numpy files, includes commonly used algorithms for data processing and it should expose a C++ and a Python API.

In addition, code should be well tested, documented, and should include logging and error handling. The library should be designed to be extensible, so that new features can be added easily in the future. and also flexible so that it accomodates different and upcoming use cases.

The library should be designed to be efficient, so that it can process data in real-time, and should be able to handle large amounts of data. It should use parallelism to distribute the load on multiple cores, and should be able to take advantage of the GPU for processing data. On the other hand it should abstract the low level details of the hardware and network communication to make it easy to use for the scientists.

5 Work Methodology

We used the **Kanban** methodology to manage the project. Kanban is a subsystem of the Toyota Production System (TPS), which was created in 1940s to control inventory levels, the production and supply of components, and in some cases, raw material. [JG10]
The board is divided into several columns:

- **Backlog** Contains all the tasks that need to be done.
- **To Do** Contains the tasks that are ready to be worked on.
- **In Progress** Contains the tasks that are currently being worked on.
- **Done** Contains the tasks that are completed.

5.1 Tools

We used Github Projects to manage the Kanban board. Github Projects is a tool that allows you to create a Kanban board and manage your tasks. It is integrated with Github, so you can link your tasks to your code, and track your progress easily. We also used Github Issues to create tasks, and Github Pull Requests to review and merge the code.
The Kanban boards were available for all the group members (involved in project or not), so that they can see the progress of the project, and contribute to it if needed. The boards were updated regularly, and the progress was tracked using the boards. The boards were also used to plan the work, and to assign tasks to the group members.

5.2 Benefits of Kanban

The Kanban methodology has several benefits:

- **Visibility** The Kanban board provides a visual representation of the work that needs to be done, and the progress that has been made.
- **Flexibility** The Kanban board is flexible, and can be easily adapted to the needs of the project.
- **Efficiency** The Kanban board helps to prioritize the work, and to focus on the most important tasks.
- **Collaboration** The Kanban board is a collaborative tool, and can be used by all the group members to track the progress of the project.

5.3 Meetings

We had regular meetings with the group members to discuss the progress of the project, and to plan the work. On each Tuesday, The whole group meets for about an hour to discuss the progress of the multiple projects that are being worked on. This meeting helps to keep everyone informed about the progress of the projects, and to identify any issues that need to be addressed.
In addition, on each Friday, a one-on-one meeting is held with the project supervisor to discuss the progress made during the week, and to plan the work for the next week. This is a more detailed meeting where we discuss the tasks that need to be done, and the kep track of the progress of the project.
Furthermore, the group has an open door policy, where anyone can ask for help, or discuss any issues that they are facing. Knowledge sharing is encouraged, and regular short discussions help overcoming the roadblocks that one might encounter.

6 High Level Planning

Even though the work methodology is highly flexible, we have a high level planning that we follow. The project is divided into several phases:

- **Phase 1: Research and Project Setup** In this phase, we researched the existing solutions, identified the limitations of the existing software, setup the project, and developed a high level architecture for the new library.
- **Phase 2: Implementation of the File Module** In this phase, we implemented the file IO module, which is responsible for reading and writing raw data files and numpy files.
- **Phase 3: Implementation of the Network Module** In this phase, we implemented the network module, which is responsible for acquiring data from receiver servers, and streaming data.
- **Phase 4: Implementation of the Processing Module** In this phase, we implemented the processing module, which includes commonly used algorithms for data processing.
- **Phase 5: Implementation of the Python API** In this phase, we implemented the Python API, which allows users to use the library from Python.
- **Phase 6: Documentation, Testing and Evaluation** In this phase, we tested the library more thoroughly, evaluated the performance, documented the code and added tutorials.

It is important to note that the phases are not fixed, and can be adapted to the needs of the project. The phases are used to plan the work, and to track the progress of the project. For example the python API was implemented in parallel with the other modules, as it was very useful to test the library and to get feedback from the users.

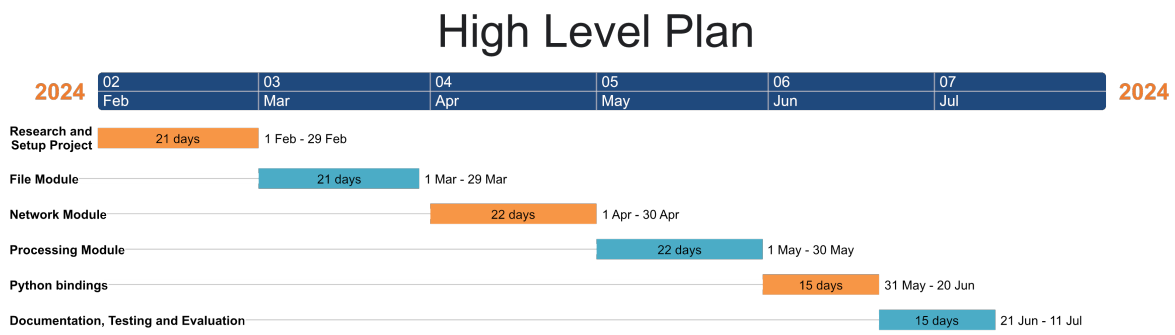


Figure I.3 – Gantt Diagram of the 6 phases of development. A margin was left at the end of the project to account for holidays, vacation days and development delays

Conclusion

In this chapter we presented the host company and the key difficulties faced by its detectors group. Following this we established the goals of the project and the methodology that we will follow to achieve them.

Part II

Chapter 2

Chapter II

Requirement Specification and Overall Architecture

Summary

Introduction

The requirement specification is the first step in the software development process. It is the basis for the design and implementation of the software system. It is the process of defin-

ing, documenting and maintaining the requirements of the system. The requirements are the description of the system services and constraints that are to be implemented.
In this chapter we will present the requirement specification of the project, we will define our actors, the functional and non-functional requirements of the system. We will also present the overall architecture of the project and the design principles that we will follow.

1 Requirement Specification

1.1 Actors Identification

Our library is developed for one main actor: the scientist.
The scientist is the person who is going to use the library to develop new data processing algorithms, acquire data from different IO sources, analyze the data and visualize the results. It is assumed that the scientist is very competent but is not necessarily an expert in software development. The library is designed to be easy to use and to provide a high level of abstraction to the scientist.

1.2 Functional Specification

1.2.1 File Input/Output

The library should provide the scientists an interface to handle file input/output:

- Read/Write frame data from different file formats: Raw, JSON, HDF5 and Numpy.
- Reorder the binary frame data according to the detector type.
- In case of part files (data split accross multiple files), the library should be able to read all the files and reconstruct the data.
- Read/Write metadata from different file formats: JSON, Raw.
- Define a new cluster file format (.clust2) to store cluster data. The file format must be flexible to accomodate different types of structured data.
- Read/Write from the new cluster file format.

1.2.2 Network Input/Output

The library should provide the scientists an interface to handle network input/output:

- Read frame data from multiple receivers using ZeroMQ. [Hin13]
- Synchronize the data streams coming from different receivers and merge them into a single data stream.
- Send data to multiple server using ZeroMQ. Thses servers can be other existing GUI applications.
- Provide an interface to distribute tasks accross multiple workers using ZeroMQ.

1.2.3 Data Processing

Although the task of data processing is very specific to the scientist, the library should provide a set of basic data processing algorithms that can be used as building blocks for more complex algorithms. In addition implementing these basic algorithms will help the scientist to understand the library and how to use it. Therefore helps in acclerating the adoption of the library.
Specifically the library should implement a cluster finder algorithm that can be used to find clusters in a frame of data. and also a pedestal subtraction algorithm that can be used to remove the background noise from the data.

1.2.4 Python Interface

Most scientists prefer to use Python for their data analysis tasks. Python has become the de facto language for data analysis and machine learning. It is easier to use than C/C++ and is less verbose. It has a large number of libraries that can be used for data analysis and visualizations.
Therefore the library should provide a Python interface that can be used to access its functionalities. The Python interface should be easy to use, similar to other Python libraries like Numpy and Pandas.

1.3 Non-Functional Specification

1.3.1 Performance

The library is expected to handle around 10GB/s of data streaming from the network and process it in real time. In addition processing large data files should be done as fast as possible

to allow the scientist to iterate quickly on their work. These are very challenging requirements and will require a lot of optimization in all levels of the library. The library should be able to run on a single CPU core and also on multiple CPU cores to parallelize the processing.

1.3.2 Reliability

The library should be able to handle different types of errors that can occur during the data processing. It should be able to recover from these errors and continue processing the data. The library should also be able to handle different types of data corruptions that can occur during the data acquisition. For example in case of a network error, the software should reconnect to the network and continue receiving the data. Furthermore, Thorough testing on multiple levels and on different scenarios is required to ensure the integrity of the code.

1.3.3 Portability

Nothing is assumed about the platform on which the library will run. It must be able to run on different operating systems (Windows, Linux, MacOS). and on different hardware architectures (x86, ARM). It cannot rely on specific kernel features or syscalls that are not available on all platforms.

1.3.4 Extensibility

Use cases and requirements can change over time. The library should be able to adapt to these changes. The software design must allow developers and scientists to add new functionalities easily. Careful considerations must be taken to make easy to add features without breaking the existing code.

1.3.5 Maintainability

The development of the library is expected to continue for a long time. The library should be easy to maintain and evolve. The code should be well documented and easy to read. Explanations should be provided for complex algorithms and data structures. Helpful documents should be provided to help new developers understand the code and how to contribute to the project.

1.4 Requirements Specification Analysis

First, given the harsh performance requirements, the library should be implemented in a compiled language. Rust, C++, Java are possible candidates for this task. Rust is a new language

that is gaining popularity for systems programming. It is designed to be safe and fast [BA22]. However, it is still a young language and the ecosystem is not as mature as C++. Java is a good candidate for performance and portability, but it is not as fast as C++. Therefore, C++ is the best candidate for this task. It is a mature language with a large ecosystem and a large number of libraries that can be used. It is also a very fast language that can be optimized to run close to the hardware. Plus, it is used in many related scientific solutions.

Second, to provide high performance python code, the library should be implemented in C++ and then wrapped in Python using the Pybind11 library [JRM17]. Pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code. It is easy to use and provides a high level of abstraction to the developer. It is also very fast and does not introduce a lot of overhead.

Third, it is possible to modularize the library into different modules where each module is responsible for a specific domain. This will help in organizing the code and make it easier to maintain and evolve. It will also help in parallelizing the development of the library. The library can be divided into the following modules:

- core module: responsible for the basic data structures that will be used in all other modules.
- file_io module: responsible for reading and writing data from different file formats.
- network_io module: responsible for reading and writing data from the network.
- processing module: responsible for implementing the data processing algorithms.
- python module: responsible for exposing the library to Python.

Fourth, for more flexibility and extensibility, the library should be implemented using the Clean Architecture and the SOLID principles.
Finally, extensive testing is required to ensure the reliability of the library. Unit tests, integration tests and end to end tests are all required. In addition to that, the library should be tested on different platforms and different hardware architectures to ensure its portability.

2 Overall Architecture and Guidelines

Given the requirements of the project, software architecture and design principles must be carefully chosen to ensure the success of the project. In this section we will present some of the

design principles and architectural patterns that we will follow in the project.

A software solution should provide to stakeholders two main things: behavior and structure. The behavior is the functionality of the system, what the system does. It answers for the functional and non-functional requirements of the system. The structure is the organization of the system, how the components of the system are organized and how they interact with each other. A good structure is essential for the maintainability and extensibility of the system.

Investing time in the design of the software architecture is very important. Specifications can change, requirements can change, but the architecture is the foundation of the system and it should be solid and flexible enough to accommodate these changes.

> The goal of software architecture is to minimize the human resources required to build and maintain the required system
>
> Robert C. Martin [Martin17]

As the quote by "Uncle Bob" suggests, the goal of software architecture is to reduce the software development costs in the project's early life. But also it is crucial to bring down the cost of new features and maintenance as project evolves and becomes more complex.

Several architectural patterns exist such as: Layered Architecture, Hexagonal Architecture [FP09], Clean Architecture [Martin17], Onion Architecture [Palermo08], DCI [CR14] etc. Although these patterns differ they share some common principles such as separation of concerns, testability, and independence from frameworks, databases, and UIs.

In this project we will follow the Clean Architecture.

2.1 Clean Architecture

The clean architecture was first introduced by Robert C. Martin (also known as Uncle Bob) in his blog post [MarBlog12] and later in his book "Clean Architecture: A Craftsman's Guide to Software Structure and Design" [Martin17]. Like other architectural patterns, the clean architecture promotes separation of concerns, testability, and independence from frameworks, databases, and UIs.
As shown in figure II.1, the clean architecture is composed of several layers:

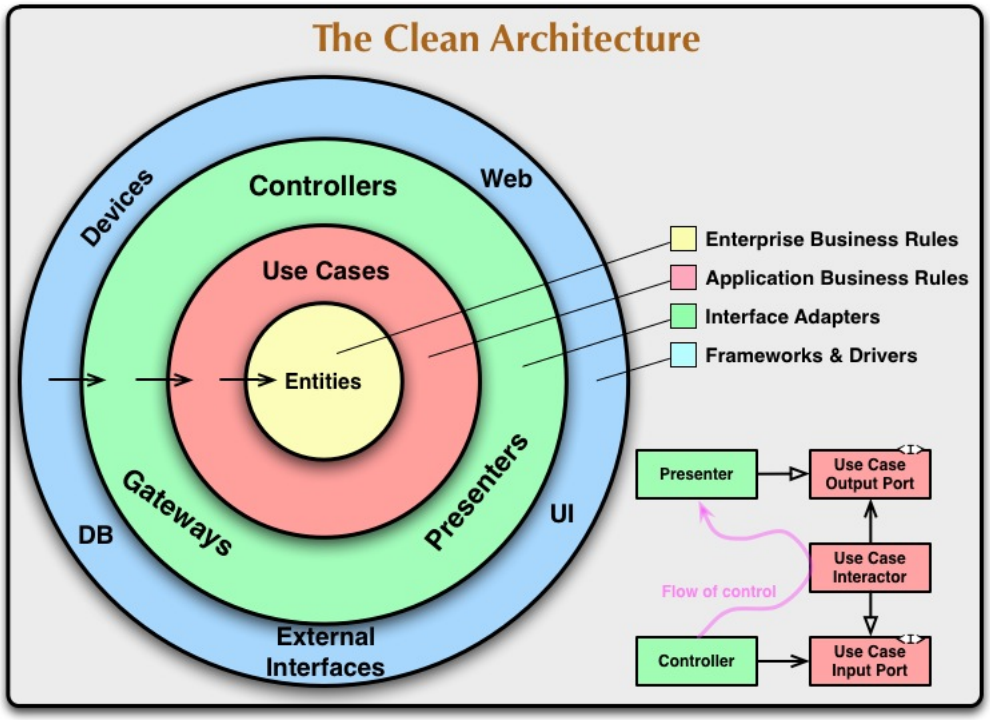


Figure II.1 – Diagram of the Clean Architecture: the circles represent the different layers of the architecture, the arrows represent the dependencies between the layers. The bottom right part of the diagram shows how dependencies are inverted.

- Entities: This layer contains the business objects of the system. These objects are the core of the system and they are independent of any other layer.
- Use Cases: This layer contains the application-specific business rules. It contains the use cases of the system. Each use case is a single action that the system can perform.
- Interface Adapters: This layer contains the interfaces that the system uses to communicate with the outside world. It contains the presenters, controllers, and gateways.
- Frameworks and Drivers: This layer contains the frameworks and drivers that the system uses. It contains the web frameworks, databases, and other external systems.

The clean architecture promotes the separation of concerns and the independence of the layers. Each layer is independent of the other layers and can be replaced without affecting the other layers. It is very important that the inner circles do not depend on the outer circles. The inner circles will not know anything about functions, classes, or modules in the outer circles. This known as the **Dependency Rule**.

> "This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle." [MarBlog12]

For communication between the layers, interfaces are used. The inner layers define interfaces that the outer layers must implement or call. The dependency inversion principle (will be explained in the next section) is used to decouple the layers and make them independent of each other.

2.2 SOLID Principles

The SOLID principles are a set of five principles that are used to design object-oriented software. They were introduced by Robert C. Martin in his paper "Design Principles and Design Patterns" [Mar00].

2.2.1 Single Responsibility Principle (SRP)

The SRP states that:

> "A module should have only one reason to change." [Martin17]

This means that a class should have only one responsibility and serves one actor. If a class has more than one responsibility, it should be split into multiple classes. This will make the code easier to understand and maintain.

2.2.2 Open/Closed Principle (OCP)

The OCP states that:

> "A software artifact should be open for extension but closed for modification." [Mey97]

This means that the code should be designed in such a way that its features can be extended without the need to modify the existing code. This can be acheived by a plethora of design patterns and object oriented principles.

2.2.3 Liskov Substitution Principle (LSP)

As Barbara Liskov defined it in 1988:

> "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T" [Lis87]

Or in simpler terms: Subtypes must be substitutable for their base types. In the case of classes this means that a subclass should be able to replace its parent class without affecting the correctness or behavior of the program.

2.2.4 Interface Segregation Principle (ISP)

The ISP states that:

> "Clients should not be forced to depend on methods that they do not use." [Mar03]

This means that interfaces should be small and focused. The ISP combats the problem of interface pollution. When classes depend on an interface it does not fully use, it is forced to implement or depend on methods that are not needed. This can lead to unnecessary dependencies and coupling between classes. and code might break when the interface is updated.
The solution proposed by the ISP is to split the interfaces into smaller interfaces that are more focused. This way the classes can depend on the interfaces that they need and not on the "fat" interface.

2.2.5 Dependency Inversion Principle (DIP)

The DIP is defined in the Agile Software Development book by Robert C. Martin [Mar03] as:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

The dependency inversion principle is crucial for the successful implementation of the clean architecture. It is used to decouple the layers and make them independent of each other. The DIP is the mechanism that allows the inner layers to define interfaces that the outer layers must implement or call. This way the inner layers do not depend on the outer layers and can be replaced without affecting the other layers.

2.3 C++ Specific Design

The library will be implemented in C++ and will be wrapped in Python. For this reason some specific design considerations must be taken into account. C++ is a very powerful language that provides a lot of features that can be used to implement almost any design. However, it is also a very complex language that has its own pitfalls and limitations.

2.3.1 Templates Metaprogramming

C++ provides a powerful feature called templates that can be used to implement metaprogramming. Metaprogramming is the writing of programs that write or manipulate other programs. Templates allows the programmer to write generic code that will be instantiated at compile time. This can be used to implement generic algorithms that can work with different types of data. Templates can make the code much more efficient as it moves the computation from runtime to compile time.

Templates are a core design feature of C++. They are used extensively in the C++ standard library and in many other libraries. However, templates can be very complex and can lead to very long compile times and cryptic compile error messages.

From a software design perspective, being aware of templates and how to use them is very important. Templates can be used for static polymorphism, to implement generic algorithms, and to implement compile-time checks.

The very important constraint to keep in mind for this project is that the C++ library will be wrapped in Python. In practice this is done by shared libraries that are loaded by the Python interpreter. Hence, the C++ code must be compiled first before it can be used in Python.

Listing II.1 – Example of a Templated Function

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

In the above example II.1, the function add is a templated function that takes a type T and an integer n as template parameters. The function adds n to the input parameter a and returns the result.

In C++ code whenever we call this function with a different set of parameters the compiler will generate a new version of the function. However, in Python we cannot do this as the interpreter will not be able to generate new versions of the function at runtime.

Therefore, the use of templates in the C++ code must be limited to the minimum and only used when it is really necessary. The C++ code should be as generic as possible but not at the expense of the Python interface.

In some use cases we are able to instantiate the C++ templated code for a set of types or values that the function expects. For example we can define function for numerical types (int8, int16, uint8, uint16, float, double ...) However we should always be aware that this can lead to code bloat and increase the size of the shared library.

2.3.2 C++ Idioms

C++ is an "old" language that has evolved over the years. It offers a lot of complex features that can be combined to create very powerful and efficient code. It also has limitations especially as it is not memory safe and it is easy to introduce bugs.

For this reason, it is important to follow some idioms and best practices when writing C++ code. Some of the most important idioms are:

RAII (Resource Acquisition Is Initialization): This idiom states that important resources' lifecycle should be bound to the lifetime of an object. This way the resource is automatically acquired when the object is constructed and released when the object goes out of scope. Another name for this idiom is Scope Bound Resource Management.

One of the most common errors in C++ is allocating memory, opening files or sockets and forgetting to release them. This can lead to leaks in the program and can cause the program to crash. The RAII idiom is widely used to avoid these problems.

As this project will use a lot file pointers, sockets and memory resources, it is very important to wrap these resources in RAII objects. This will ensure that the resources are released when they are no longer needed.

Curiously Recurring Template Pattern (CRTP): The CRTP is a design pattern that is used to add functionality to a class by inheriting from a base class. The base class is a template class that takes the derived class as a template parameter. This allows the base class to access the members of the derived class.

Listing II.2 – Example of CRTP

```
template <typename T>
class Base {
    void foo() {
        static_cast<T*>(this)->bar();
    }
};

class Derived : public Base<Derived> {
    void bar() {
        // do something
    }
};
```

In the above example II.2, the class Base is a template class that takes a type T as a template parameter. The class has a function foo that calls the function bar of the derived class. The derived class must inherit from the base class and pass itself as the template parameter.

It is a technique that involves using inheritance and static polymorphism to achieve a form of compile-time polymorphism. Indeed the CRTP allows for compile-time optimization and inlining of code since the method calls are resolved at compile time. It also eliminates the runtime overhead associated with virtual function calls, making it suitable for performance-critical code.

Type Erasure Type erasure is a programming technique in C++ used to enable polymorphism without sacrificing performance or introducing unnecessary runtime overhead. It allows the creation of interfaces and containers that can work with objects of different types in a type-safe manner while deferring the determination of the actual types until runtime.

Type erasure is often used to implement generic programming in C++ where the type of the objects is not known at compile time. It is also used to implement interfaces that can work with objects of different types.

the std::function and std::any classes in the C++ standard library are examples of type erasure. They allow the creation of objects that can hold objects of different types and call their methods without knowing the actual type of the object.

Listing II.3 – Example of Type Erasure

```
Shape circle = Circle(5);
Shape square = Square(5);
std::vector<Shape> shapes;
```

In the above example II.3, the Shape class is a base class that is used to store objects of different types. The Circle and Square classes are derived from the Shape class. The vector shapes is a vector of Shape objects that can hold objects of different types.

Pimpl (Pointer to Implementation): The Pimpl idiom is used to hide the implementation details of a class from the user of the class. It is used to reduce the compile time dependencies and to hide the implementation details of the class.

The list of the mentioned idioms is not exhaustive. C++ offers a lot more idioms and best practices that can be used to write efficient and safe code.

2.4 Project Architectural Design

Having armed ourselves with various design principles and architectural patterns, we can now tackle the task of designing the architecture of the project.

2.4.1 Project Modules

Having identified the different requirements of the project, we can now divide the project into different modules. Each module will be responsible for a specific domain of the project.

- file_io module: Responsible for handling file input/output operations.
- network_io module: Responsible for handling network operations.
- processing module: Responsible for implementing the data processing algorithms.
- core module: Responsible for the basic data structures that will be shared between other modules.

- python module: Responsible for exposing the library to Python.

Each module should be independent of other modules and communicates with them through exposed interfaces. Furthermore, each module includes its own tests. This will help in isolating the tests and make it easier to test the modules. In general this approach focuses on separation of concerns and modularity.

2.4.2 Project Architecture

The project will follow the Clean Architecture. The project will be divided into different layers where the inner layers do not depend on the outer layers.

Figure II.2 is an illustration of how the project is organized. Boxes represent the modules that have been implemented by the project. Controllers and Presenters are not implemented but in future iterations they can be added to communicate with a UI or an API. The direction of the arrows represents the dependencies between the modules.

The core module is at the center of the architecture. It is the most stable part of the software and least likely to change. core module will contain data structure such as "`Frame`" and "`Cluster`" that will be used by higher level modules. The core module will also contain the interfaces that will be used by the other modules.

One level above the core module we find the file_io, network_io and processing modules. These modules contain the application-specific business rules and use cases. These modules will also depend on the core module.

In the Interfaces and Adapters layer we can find the python module. This module will be written in C++ and it will contain the bindings to expose the libraries functionalities. The python module will depend on all the modules below it. In case of future changes, this layer can contain Presenters to present the data on a GUI or in another formats and also it can contain controllers to handle requests from other servers or the Web.

Finally, the Frameworks and Drivers layer will contain the external components that the system will use. Hardware, Databases, Python libraries etc all reside in this layer. These components are subject to frequent unpredictable changes and should be isolated from the rest of the system.

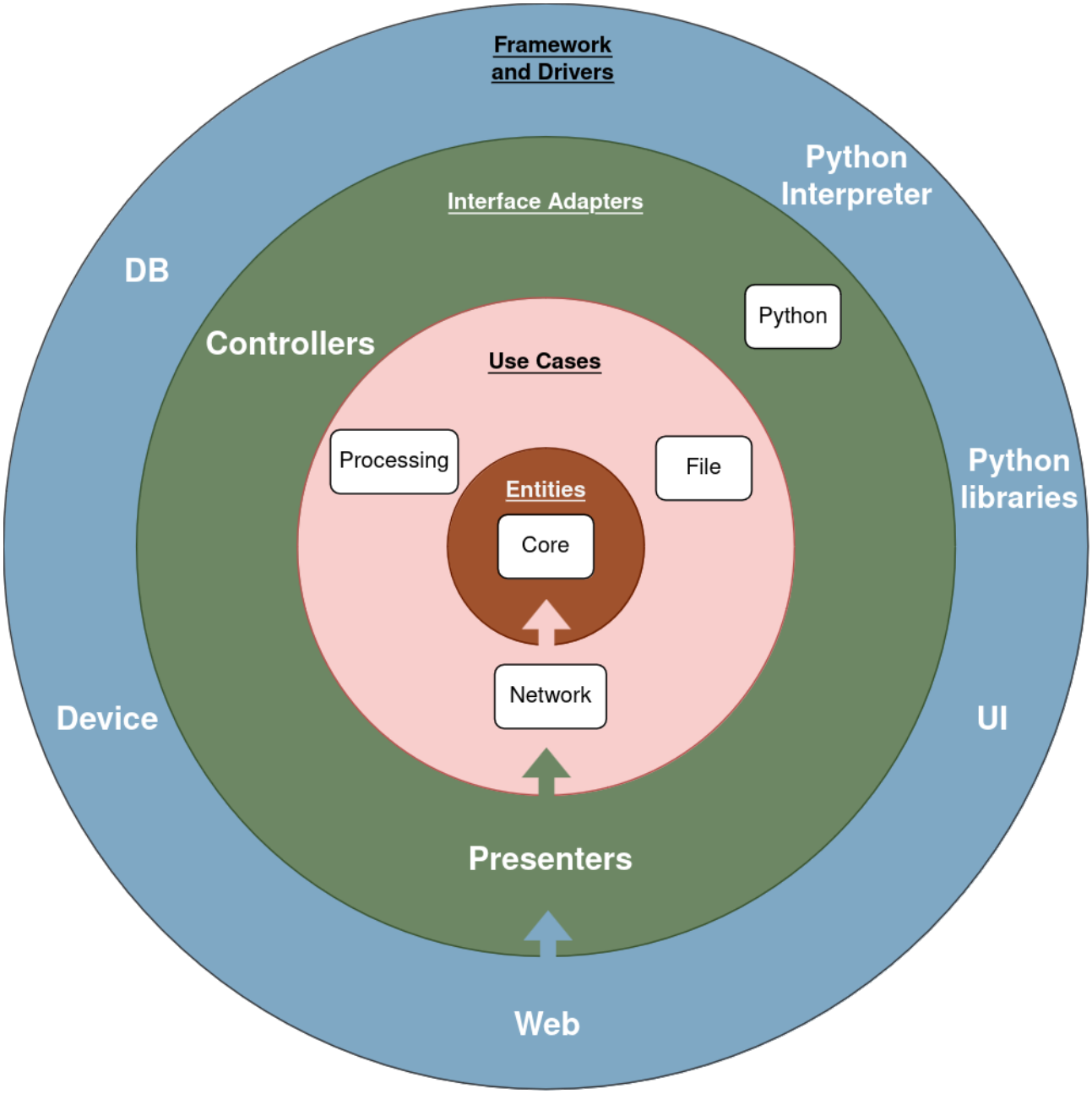


Figure II.2 – Diagram of the Project Architecture

3 Project Setup

3.1 Project Structure

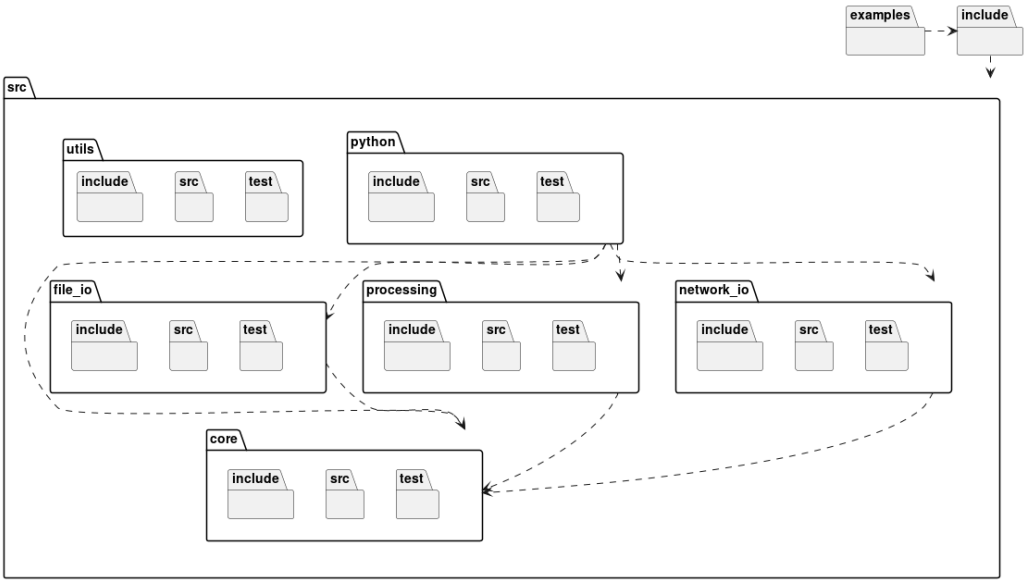


Figure II.3 – Project folder structure

The folder structure of the project is shown in Figure II.3. The file hierarchy is based on the Pitchfork project structure recommendations [Pik18]. The project is composed of a include folder containing the header files, a src folder containing the source files, an examples folder containing the examples. The root directory of the project contains a CMakeLists.txt file which is used to build the project. Each module has its own folder in the src directory. For example the core module has include, src and test folders inside it.

3.2 Build System

The project uses CMake as the build system. CMake is a cross-platform build system that generates native build files for the platform of your choice. It is a powerful tool that can be used to build, test and package software. The CMakeLists.txt file in the root directory of the project is used to configure the project. Each module also has its own CMakeLists.txt file which is used to configure the module and build the sources files and tests. CMake will compile each module independently as static libraries and link them together to create the final executables or share objects.

In this project CMake is also used to manage the dependencies. The project uses external libraries such as ZeroMQ, FMT, Catch2... CMake downloads and builds these libraries automatically. This is done using the FetchContent module in CMake. The FetchContent module allows to download and build external projects at configure time.

3.3 Version Control

The project uses Git as the version control system. Git is a distributed version control system that allows tracking changes in the source code. It is a powerful tool that allows multiple people to work on the same project at the same time.

The remote repository is hosted on GitHub. GitHub is a web-based platform that provides hosting for software development version control using Git. It also provides collaboration features such as bug tracking, feature requests, task management and wikis for every project. Pull requests are used to merge changes from one branch to another. This allows reviewers to check the code before it is merged into the main branch.

3.4 Testing and Continuous Integration

The project uses Catch2 as the testing framework. Catch2 is a modern, C++-native, header-only, test framework for unit-tests. It is easy to use and provides a lot of features such as BDD-style testing, test cases, sections, assertions, tags, test fixtures, test cases and test suites.

Each module has its own test folder containing the test files. The test files are compiled into a test executable which is run to test the module. The test executable is built using the CMakeLists.txt file in the test folder of the module. A single test executable is built for the whole library which runs all the tests of the submodules. Tags can be used to run or exclude specific groups of tests.

Continuous integration is used to automatically build and test the project. GitHub Actions is used to run the tests on each push to the repository. The tests are run on multiple platforms including Windows, Linux and MacOS. And different configurations are used such as Debug and Release, with and without Python bindings, download dependencies with CMake or use system libraries... Furthermore, the code is checked for formatting using clang-format and for linting checks using clang-tidy. Linting checks are used to find potential bugs and code smells

in the code.

Conclusion

In this chapter we oversaw the requirement specification of the project and we have defined the broad lines of the project architecture. This architectural planning will guide the development of the project and will help in organizing the code. Although it costs time and effort to design the architecture of the project, it is a long term investment that will pay off in the future. The architecture will help in maintaining the code, adding new features and fixing bugs. It will also help in understanding the code and in onboarding new developers.

Part III

Chapter 3

Chapter III

Implementation of a Flexible Data Analysis Library for Hybrid X-ray Particle Detectors

Summary

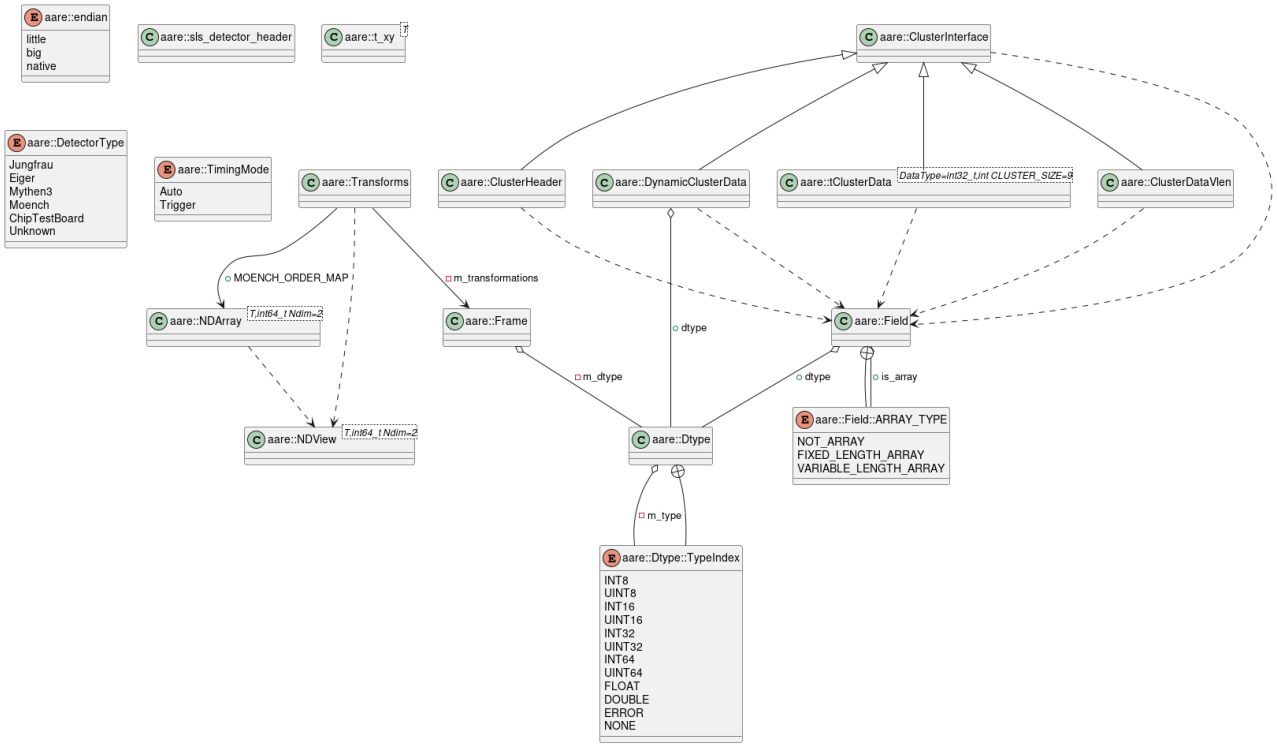


Figure III.1 – Core module simplified class diagram

Introduction

In this chapter we will go through the implementation and design choices for each module.

1 Core Module

The core module is the base module of the library. It contains the core classes and functions that are used by the other modules. It is crucial that the core module is well designed and focus on stability, performance and extensibility. From the requirement specifications we can identify that the file_io, network_io and processing modules will need to use a `Frame` and a `Cluster` class. The `Frame` class will contain the data sent by the detector and the `Cluster` class will contain the data after processing. Hence the core module will define these two classes/interfaces.

1.1 Class Diagram

Figure III.1 shows the class diagram of the core module. The attributes and methods of the classes are not shown in the diagram for simplicity.

1.2 Dtype Class

The library will need to read write data from multiple streams such as files, network, memory... The library will also handle different data types such as integers, floats... The Dtype class is used to represent the data type of the handeled stream of bytes. The Dtype class also define the endianness of the data. To represent the data types, this class internally uses an enum class. String representation can also be used to create a Dtype object. The string representation is defined as "<" or >" for little or big endian followed by the data type and its size in bytes. For example "<i4" represents a little endian int32 and ">f8" represents a big endian float64.

The Dtype class is also very important for transferring data between the C++ and Python bindings. The Python bindings will use the Numpy library to handle the data. Numpy uses the python standard of buffer protocol[Fou22] to handle data exchange. The buffer protocol is a way to expose the data of an object to other objects. It is useful when we don't want to copy the data but just share it. The library allows data to go from C++ to Python and vice versa with or without copying. The Dtype class is used to define the data type of the incoming of outgoing buffer.

1.3 Frame Class

The `Frame` class contains the data of a detector "image". The data is stored in a buffer of bytes. and it uses Dtype to define the data type of the buffer. The `Frame` class also contains the shape of the data.

`Frame` objects will be used across the library. The file_io module will use `Frame` objects to read and write data from files. The network_io module will use `Frame` objects to send and receive data over the network. The processing module will use `Frame` objects to process its data. Therefore it is very important that the `Frame` class is as simple as possible and as general as possible to be used by all the modules.

In the first iterations of the library, the `Frame` class was templated on the type of the data. This allowed the user to manipulate the data of the frame directly e.g. `Frame<int32_t>[x][y] = value` . However, this is now considered a bad design choice. Templating `Frame` forced the user to know the type of the data before creating a `Frame` object. It also propoagated complexity to higher level layers. For example, file_io functions that read data from a file and returned Frame objects also had to be templated and so on. This is why the `Frame` class is now a simple class

untemplated on the data type. and this point is why NDView and NDArray were introduced.

1.4 NDView and NDArray

The NDView and NDArray classes are used to manipulate the data of a Frame object or other buffers. They stand for N-Dimensional View and N-Dimensional Array. The NDView class is a non-owning view of the data. It is used to access the data of a buffer without copying it. It can also be used to reshape (in numpy terms) the data. NDArray is an owning view of the data. It copies the data of the buffer and owns it.

The NDView and NDArray classes are very helpful for processing the data. They are inspired by the Numpy library which is a powerful library for numerical computing in Python [Har+20]. The Numpy library had a huge influence on the development of python libraries and it is widely used accross different fields. Users of this library can reuse their knowledge of Numpy and apply it to this library. This helps to reduce the learning curve of the library and accelerates its adoption. In addition, When the library receives a numpy object from python it can be converted to a NDView/NDArray object and vice versa.

1.5 Cluster Classes

We refer to the group of classes that handle cluster data the Cluster classes. The Cluster classes have gone through multiple iterations. From working around the limits C++ and Python bindings to balancing the tradeoffs of performance vs flexibility vs ease of use. This has led us to the current design of the Cluster classes.

After receiving frames from disk or the network a common way to process the data is to find the photon of electron clusters in the image. A cluster is a group of pixels that are close to each other and have a high intensity. The cluster finding algorithm will be part of the processing module and is beyond the scope of the core module. Clusters need to be generated, saved and read from files and in future iterations sent over the network. This is why a unified Cluster class definition is needed in the core module.

Clusters are represented as a list of pixels. There's no single way to represent a cluster. The group currently uses multiple forms of clusters such as: fixed length clusters which are a list of pixels of a fixed size (3x3 or 2x2) centered around the maximum pixel. and variable length clusters which are a list of pixels that are adjacent to each other. Different cluster formats have

different uses and applications. For example fixed size clusters are more commonly used for photon clusters while variable size clusters are more commonly used electron clusters as they leave a trail of pixels.

Given these requirements, First an interface class was created to set the define the methods that a Cluster class should have. This interface class is only used to inherit its methods in case the subsclass does not implement them. It should be noted that the interface class does not contain any virtual methods and is not meant to be used as a polymorphic class. Any implementation of a Cluster class should inherit from the ClusterInterface class. Although it isn't implemented, the ClusterInterface can use the Curiously Recurring Template Pattern (CRTP) and Substitution failure is not an error (SFINAE) idioms to enforce that the subclass implements the methods in compile time.

First we will explain the general design of a Cluster class. The structure of a Cluster object is defined by a list of Field objects. A Field object is a simple class that contains a label (`std::string`), a dtype (`Dtype`) an enum is_array that defines if the field is a scalar or fixed length array or a variable length array and finally a size attribute that is used to indicate the size of the fixed length array.
Example III.1 the definition of the structure of a Cluster object to represent a 3x3 int32 cluster that also contains an x and a y attributes:

Listing III.1 – Example: Definition of a Cluster structure

```
// cluster definition
struct Cluster{
    int32_t x;
    int32_t y;
    int32_t pixels[9];
};

// cluster fields description
std::vector<Field> fields = {
    Field("x", Dtype("<i4"), Field::Scalar),
    Field("y", Dtype("<i4"), Field::Scalar),
    Field("pixels", Dtype("<i4"), Field::FixedLengthArray, 9)
};
```

Given that we can define the structure of a Cluster object, we can define a DynamicCluster class that holdes the data of any cluster and interprets it according to the structure. The DynamicCluster offers set and get methods to manipulate and access its data.

Although the DynamicCluster class is very flexible and can be used to represent any type of cluster, it is not the most efficient way to represent all clusters. DynamicCluster allocates its data on the heap and its contents is spread accross the memory. For fixed size clusters this is not efficient as the data can simply be represented in a struct and be stored in contiguous memory.
Example III.2 shows how to read multiple clusters from a file directly to memory. This is not possible with the DynamicCluster class. This is why `tCluserData<typename DataType,int CLUSTER_SIZE>` has been introduced.
The tClusterData class is a templated class that represents a fixed size cluster. The data will be stored contiguously in stack memory. The tClusterData class is more efficient than the DynamicCluster class but it is less flexible.

Listing III.2 – Example: Reading multiple clusters from a file

```
struct Cluster {
    int32_t x;
    int32_t y;
    int32_t pixels[9];
}; // 3x3 cluster
std::vector<Cluster> clusters(10); // list of 10 clusters stored contiguously
    in memory
fread(clusters.data(), sizeof(Cluster), clusters.size(), file); // directly
   read 10 Clusters to memory
```

These were tha main components of the core module. It also includes other classes and enums that are less important but are used by the other modules.

2 File IO Module Implementation

In this section we will go through the design and implementation of the file_io module. The file_io module is in general responsible for storing data structures and reading them from disk. Specifically it should offer features to read and write Frame and Cluster objects to the disk.

Figure III.2 shows the class diagram of the file_io module. The module is composed of two parts for handling Cluster and Frame objects.

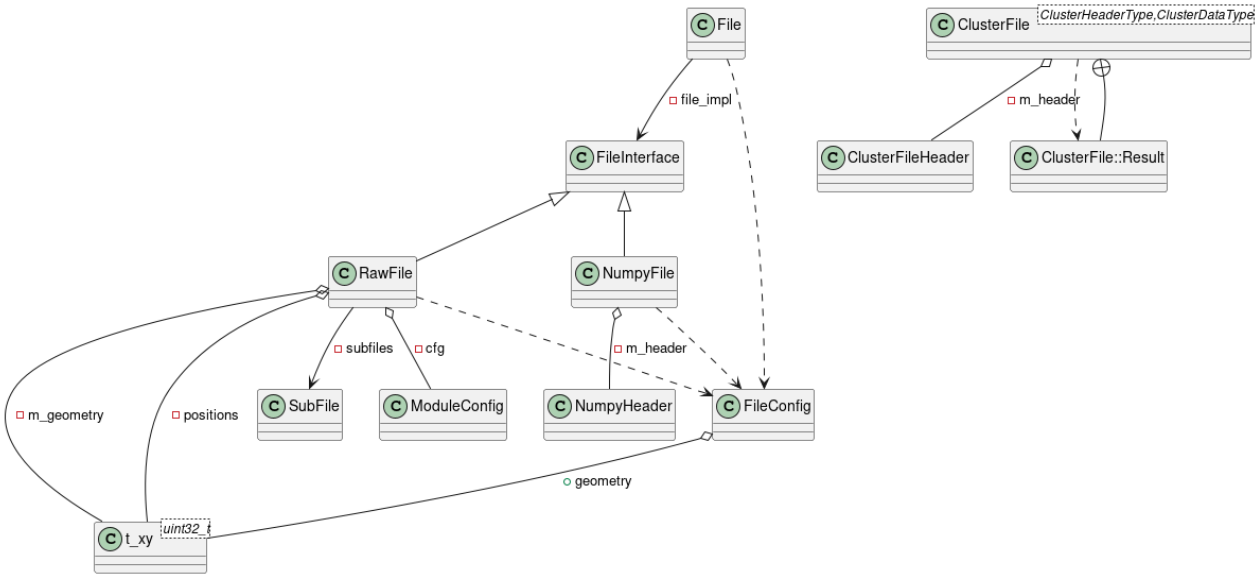


Figure III.2 – File IO module class diagram

2.1 Frame IO

Reading and writings Frame objects to disk is a central feature of the library. The library should focus on performance and flexibility when reading and writing Frame objects. Performance is important as the library should be able to read and write huge amounts of data quickly. Flexibility is important because different file formats can be used to store the data e.g. Binary, HDF5, Numpy...

From the existing slsDetectorPackage library, frame data is stored in a binary format. A master file written in JSON contains the metadata of the data and a .raw file contains the data itself. For compatibility reasons with an old master file format, the library also must handle master files written in text format following a predefined structure.
In addition, since many users of the library will be using Python, the C++ library should read and write Numpy files. A Numpy file is composed of a metadata header in plain text and a binary part that contains the data.

As seen in the class diagram III.2, the `FileInterface` class is the base class for `NumpyFile` and `RawFile`. `FileInterface` is an abstract class that only has pure virtual methods. A pointer to `FileInterface` can point to either a `NumpyFile` or a `RawFile` object. This allows the user to use the same interface to read and write different file formats.
The interface offers functions such similar to the familiar C file API such as `read()` to read a

Frame, `read(int n)` to read n Frames, `seek(int n)` to move the file pointer to the n-th Frame, `write(Frame& frame)` to write a Frame.

To use C++'s inheritance and polymorphism features, pointers to the base class are used (e.g. `FileInterface* file = new NumpyFile()`). However, forcing the user to use pointers is not very user-friendly and can lead to memory leaks. This is why a `File` class following RAII idiom is introduced. It is a simple class that contains a pointer to the base class and manages the memory of the pointer. The user can use the `File` class to read and write files without worrying about memory management.

Furthermore, to optimize the reading and writing of `Frame` objects, the library implements the appropriate copy and move constructors and assignment operators. This avoids unnecessary copying of the data and improves the performance of the library.

Listing III.3 – Example: Reading and writing a Frame from a file

```
// reading a Frame from a file
File file1("data.npy");
Frame frame = file.read();

// writing a Frame to a file
auto dtype = Dtype("<u4"); // unsigned int32
FileConfig const config = {dtype, 100, 100}; // File holds 100x100 Frames of
    uint32_t
File file2(path, "w", config); // open file in write mode
file2.write(frame); // write the frame to the file
// file is closed when it goes out of scope
```

Example III.3 shows how to read and write a Frame from a file. The `File` recognizes the file format from the extension of the file and creates the appropriate object. Here in this example, a `NumpyFile` object will be constructed. Following RAII idiom, the file is closed and memory is freed when the `File` object goes out of scope. As seen from this example the `File` abstracts a lot of the complexity of reading and writing files.

2.2 Cluster IO

The Cluster IO part of the file_io module is responsible for reading and writing Cluster objects to disk. The Cluster part of the file_io module went through multiple iterations. The first

iteration handeled fixed size clusters only. This, of course, needed to change after the introduction of the DynamicCluster class. Currently, the module is implemented to handle fixed size, variable size and upcoming custom clusters.

A new file format has been introduced to store clusters and to unify the storage of different types of clusters. The file format is composed of a FileHeader written in plain text and a binary data part that contains the clusters. The header contains the metadata of the file such as the version, the number of records stored, the structure of the clusters... The header is written in plain text JSON format to be human-readable.
The binary data part contains the records. A record is a ClusterHeader followed by N-Clusters. A ClusterHeader contains the metadata of the record such as the number of clusters, frame number...
The structure of the ClusterHeader and ClusterData are defined in the FileHeader.

The library allows users to define their own cluster structures. Whether it is a custom ClusterHeader or a custom ClusterData, ClusterFile will be able to read and write it. It is also very important to manage the tradeoffs of this flexibility and performance requirements. Unlike dynamic languages such as Python or Javascript, C++ is a statically typed language. This means that the library must know the structure of the clusters at compile time. Allowing the user to define their own cluster structures can lead to a lot of complexity. This is why it is important to find a balance between flexibility and performance.

Having noted some of the limitations of C++, some of them can be overcome by using templates. The ClusterFile class is templated on the ClusterHeader and ClusterData types. This allows the user to define their own ClusterHeader and ClusterData types that follow the ClusterInterface. The ClusterFile class will be able to read and write these clusters. ClusterFile also checks if the structure's data are stored contiguously in memory. If they are, the library can read and write clusters from disk with one system call. This is much faster than reading and writing clusters one by one.

Listing III.4 – Example: Reading and writing a Cluster from a file

```
// reading a Cluster from a file
ClusterFile<ClusterHeader, ClusterData> file("data.clust2");
auto result = file.read(); // read 1 record from file
ClusterHeader header = result.header;
std::vector<ClusterData> clusters = result.data;
```

```
// writing a Cluster to a file
 ClusterFileHeader header;
// define the ClusterHeader fields
header.header_fields.push_back({"frame_number", Dtype::INT32, Field::
    NOT_ARRAY});
header.header_fields.push_back({"n_clusters", Dtype::INT32, Field::NOT_ARRAY
    });

// define the ClusterData fields
header.data_fields.push_back({"x", Dtype::INT16, Field::NOT_ARRAY});
header.data_fields.push_back({"y", Dtype::INT16, Field::NOT_ARRAY});
header.data_fields.push_back({"pixels", Dtype::INT32, Field::
    FIXED_LENGTH_ARRAY, 9});
ClusterFile<ClusterHeader, ClusterData> file("file.clust2", "w", header);

ClusterHeader header = {1, 10}; // 1st frame, 10 clusters
std::vector<ClusterData> clusters = {{1, 2, {1, 2, 3, 4, 5, 6, 7, 8, 9}}};
file.write(header, clusters);
```

Although the task of reading and writing user defined data types is complex, the library abstracts a lot of the complexity. The user only needs to define the structure of the clusters and the library will take care of the rest. Example III.4 shows how to read and write custom clusters from a file.

A hidden but important disadvantage of the current implementation is that the ClusterFile class is templated on the ClusterHeader and ClusterData types. In C++, this is not a problem as long as the ClusterHeader and ClusterData types are known at compile time. However, the python bindings will need to know the ClusterHeader and ClusterData types at compile time. This prohibits python users from defining their own cluster structures. Furthermore, the shared object file that is generated by the C++ library will be much larger. As it will contain all the possible ClusterHeader and ClusterData types. This is a tradeoff that the library has to make.

In conclusion, the ClusterFile class is a very powerful class that allows the user to read and write customly defined structures from disk. It is also optimized for performance. Stack based structures are read and written in bulk. Minimizing the number of OS file API calls. However, the ClusterFile class is templated on the ClusterHeader and ClusterData types. This can lead to large shared object files and prohibits the python bindings from defining their own cluster structures.

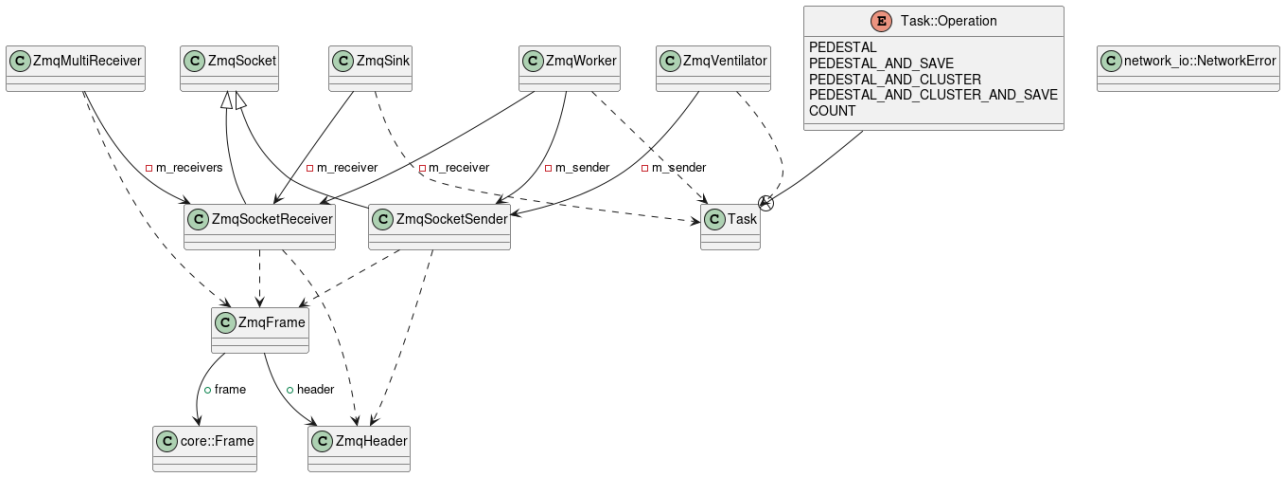


Figure III.3 – Network IO module class diagram

3 Network IO Module Implementation

The network_io module is responsible for sending and receiving Frame objects over the network. The module should interoperate with the existsing system programs such as the slsReceiver.

Figure III.3 shows the class diagram of the network_io module.

3.1 ZeroMQ

The network_io module uses ZeroMQ as the network library. ZeroMQ is a high-performance asynchronous messaging library that provides a message queue, but unlike message-oriented middleware, a ZeroMQ system can run without a dedicated message broker. ZeroMQ is a very powerful library that can be used to build complex network systems. It is also very easy to use and has bindings for many languages. [Hin13]

ZeroMQ's model provides socket communication between threads, processes and network nodes. Its sockets can work with Inter Process Communication (IPC), Transmission Control Protocol (TCP), User Datagram Protocol (UDP), WebSockets and more. ZeroMQ sockets can be connected with each other following a plethora of different patters. The most common patterns are Request-Reply, Publish-Subscribe, Push-Pull, Pair, and Router-Dealer etc.

ZeroMQ's philosophy is to provide a simple API that can be used to build complex systems. It is very flexible and extensible while also being very fast. This aligns with our library's philosophy of being easy to use, flexible and efficient.

3.2 ZmqSocket classes

As seen from the class diagram, the network_io module implements ZmqSocketReceiver and ZmqSocketSender. These classes are responsible for receiving and sending Frame objects over the network. They derive from the ZmqSocket class which is a wrapper around the ZeroMQ socket.

The ZmqSocket class follows the RAII idiom and wraps around the ZeroMQ socket. The ZmqSocketReceiver and ZmqSocketSender classes will be used by users to communicate with slsReceiver and other network nodes.

The network_io module methods return a ZmqFrame object. It is a wrapper around Frame that adds network metadata to it.

Listing III.5 – Example of using the network_io module

```
ZmqSocketReceiver socket("tcp://localhost:5555");
socket.connect(); // connect to another node
std::vector<ZmqFrame> v = socket.receive_n(); // receive frames until a stop
    signal
ZmqSocketSender socket2("tcp://localhost:5556");
socket2.bind(); // bind to a port
Frame frame(512,512,Dtype::UINT32); // create a Frame object
ZmqHeader header;
header.shape = {512,512}; // set the metadata of the frame
ZmqFrame zmq_frame = {header, frame}; // create a ZmqFrame object
socket2.send(zmq_frame); // send the frame
```

Example III.5 shows how to use the network_io module to send and receive frames. The RAII idiom as always comes in handy to disconnect sockets, free resources and close connections when the object goes out of scope.

3.3 ZmqMultiReceiver class

In several scientific experiments, multiple detectors are used to capture data. Whether it is to capture data from different angles, different parts of the sample or to balance the load, multiple detectors are used.

The ZmqMultiReceiver class is a wrapper around multiple ZmqSocketReceiver objects. It is used to receive data from multiple sources and combine them into a single stream of frames. The user of this class must specifiy the different endpoints to connect to as well as the geometry to form the frames.

The ZmqMultiReceiver handles the synchronization of the different streams. Detectors can drop frames, send them at different rates or have different delays. The ZmqMultiReceiver class must account for these issues and provide a consistent stream of frames. Fault tolerance is also important and adequate error handling and logging must be implemented.

Instead using multiple threads to listen to each detector, the ZmqMultiReceiver class uses a single thread to listen to all the detectors. It uses a poller to listen to all the sockets and receive frames. ZeroMQ simplifies the polling operation by providing a `zmq::poll()` function that can be used to poll the sockets' file descriptors. On Linux systems, `zmq::poll()` uses the kernel's `epoll` interface.
epoll is a Linux kernel system call that waits for events on file descriptors (sockets). It is used for multiplexing I/O to manage multiple connections. epoll is more efficient than the older select system call and can handle a large number of connections. Several webserver implementations such as nodejs use epoll to handle thousands of HTTP requests concurrently. [Gam+04; Mar15]

3.4 Task Distribution

The network_io module offers classes to support task distribtion over the network. Detectors are extremely fast can reach more than 10GB/s and combining multiples of them can lead to bottlenecks in network.
In addition, Some experiments can run for multiple hours and even multiple days. Storing the data on disk will become expensive overtime. 24 hours with 10GB/s leads to 864TB of data. This is why it is required to process data in real time and store only the relevant results.
For these reasons distributing the bandwidth and computation to multiple machines is the way to go.

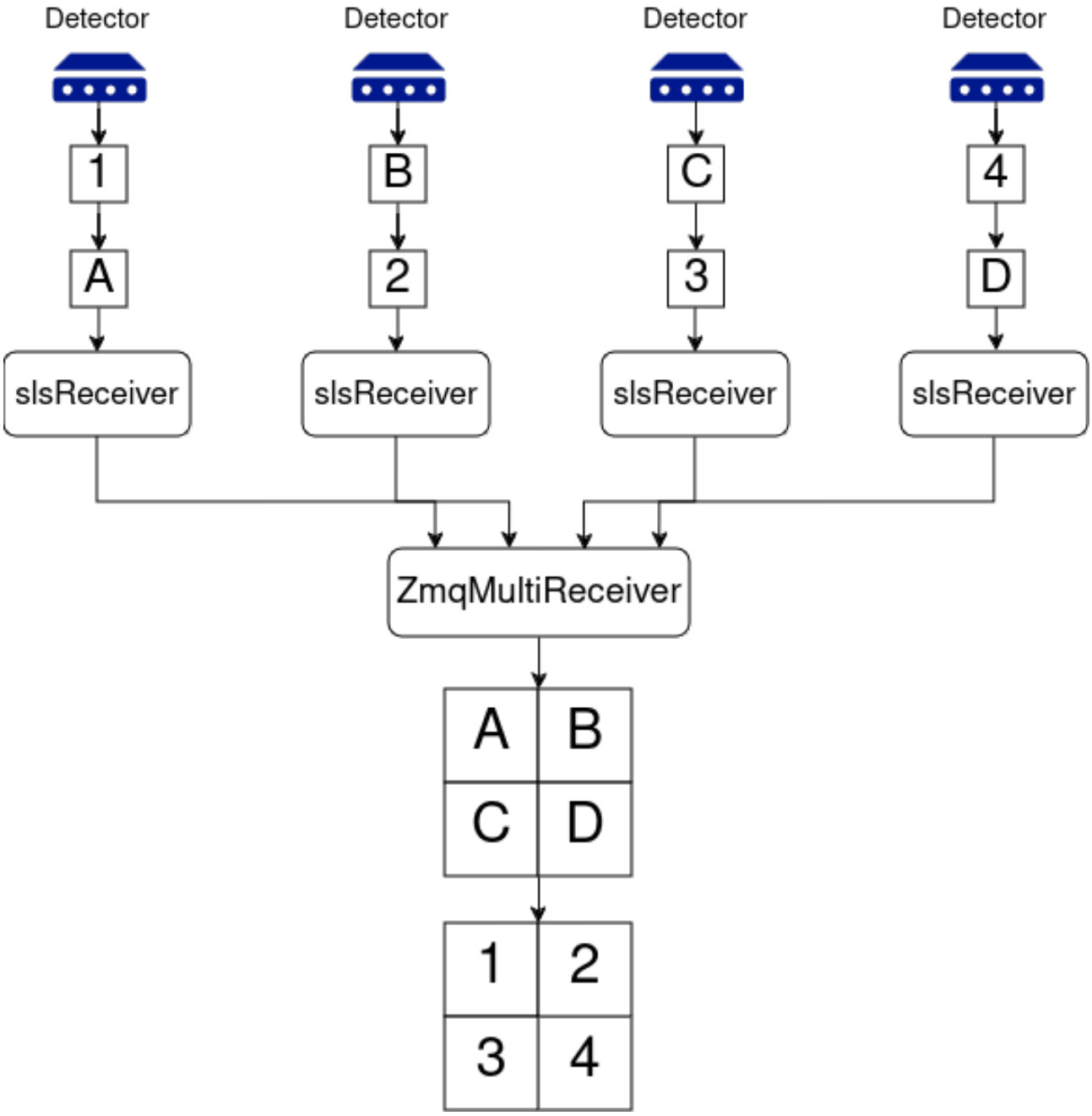


Figure III.4 – Illustration of the ZmqMultiReceiver class: multiple detectors send frames to the ZmqMultiReceiver which combines them into a single Frame. The ZmqMultiReceiver also manages the synchronization between streams.

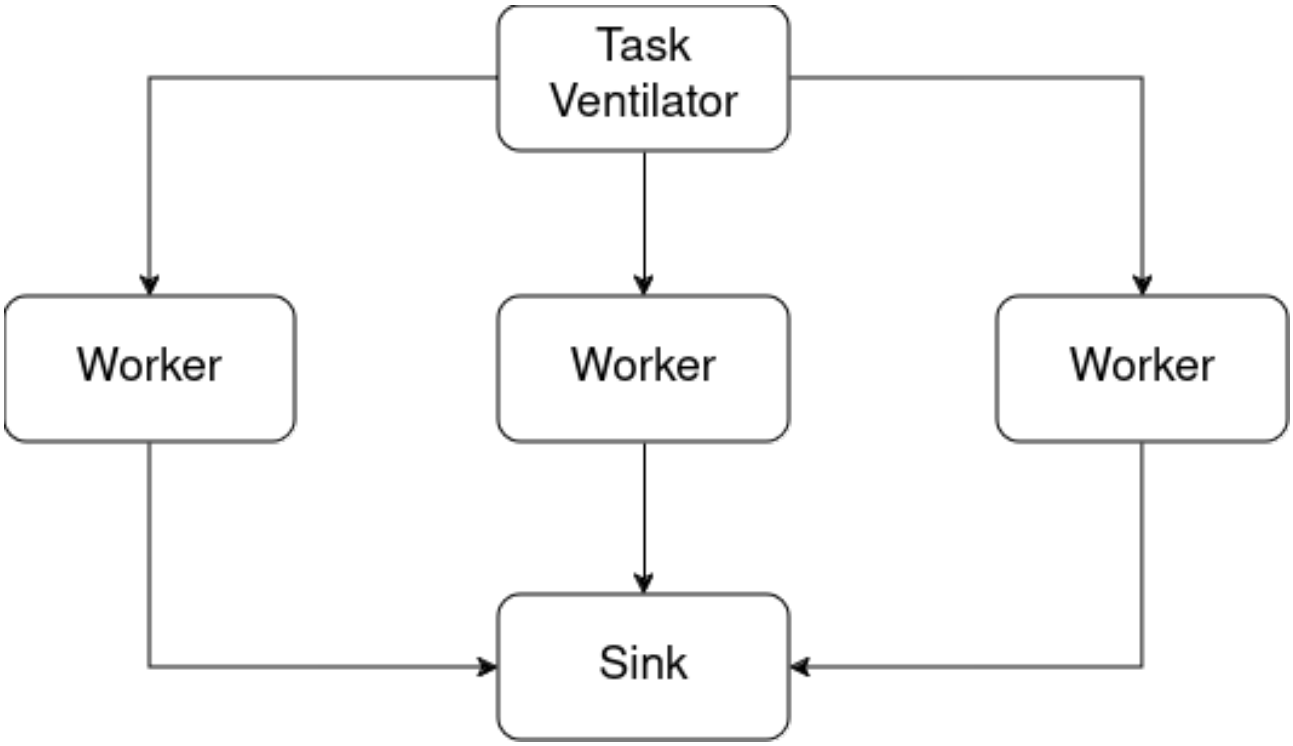


Figure III.5 – Illustration of the task distribution system. The task ventilator class distributes tasks to multiple workers. The workers process the tasks and send the results back to the sink.

Figure III.5 illustrates the task distribution system. For example the task ventilator receives frames from multiple detectors. It then creates a Task object that contains the data and specifies the steps to manipulate the data (if needed). The task ventilator distributes the tasks to multiple workers. The workers process the tasks and send the results back to the sink. The sink is used to collect the results synchronize and store them if needed.

The library provides helper classes that wrap around the ZmqSocketSender and ZmqSocketReceiver to implement the task distribution system. The `ZmqVentilator`, `ZmqWorker` and `ZmqSink` classes handle the network side of the task distribution system.

Workers connect upstream to the ventilator, this means that we can add or remove workers arbitrarily at runtime. With fair-queuing the sink collects results from the workers evenly [Hin13]. The current library does not support using multiple ventilators and sinks.

Listing III.6 – Example of a task ventilator

```
ZmqSocketReceiver receiver(endpoint);
receiver.connect();

// 1. create ZmqVentilator
ZmqVentilator ventilator("tcp://*:4321");

// 2. receive frame from slsReceiver
std::vector<ZmqFrame> zmq_frames = receiver.receive_n();

for(auto &zmq_frame : zmq_frames){
    // 3. create task
    Task task = Task::init(zmq_frame);
    task.id = zmq_frame.header.frameNumber;

    // 4. push task to ventilator
    ventilator.push(task);
}
```

Listing III.7 – Example of a worker

```
// 1. create ZmqWorker
ZmqWorker worker(ventilator_endpoint, sink_endpoint);
worker.connect();

while(true){
    // 2. receive task from ventilator
    Task task = worker.pull();

    // 3. process task
    Task result = process(task);

    // 4. send result to sink
    worker.push(result);
}
```

Listing III.8 – Example of a sink

```
// 1. create ZmqSink
ZmqSink sink("tcp://*:4322");
sink.bind();

while(true){
    // 2. receive result from worker
```

```
        Task result = sink.pull();

        // 3. store result
        store(result);
    }
```

Examples III.6, III.7 and III.8 demonstrate how to use the task distribution system. The ventilator receives frames from the slsReceiver and creates tasks. The workers process the tasks and send the results to the sink. Each example should be run in a separate process. We can scale the number of workers to match the number of available cores on a single machine. Or we can distribute the workers to multiple machines to scale horizontally.

4 Processing Module Implementation

The processing module contains the classes and algorithms used to process the data. The class diagram of the processing module is shown in Figure III.6. Currently two data processing algorithms are implemented.

4.1 Pedestal class

The pedestal algorithm is used for preprocessing the incoming frame data. Detectors have a background signal that is present even when no particles are detected. This background signal can be the still image of the sample. The Pedestal substraction algorithm is used to remove this background signal and help in the detection of key meaningful signals. The Pedestal class is used to calculate the pedestal values for each pixel. Usually the pedestal values are calculated by taking the moving average of the last N frames. However, in our case a frame of data can have 1024x1024 pixels. Keeping track of the last N frames would be memory intensive i.e $O(N)$ in memory. In order to store the last 1000 frames for the average $1024 * 1024 * 1000 * 4 = 4GB$ of memory would be needed. Instead we keep track of the sum of the last N frames and with each new frame we subtract the average of the last N frames and add the new frame.

$$\text{Sum}_i = \text{Sum}_{i-1} - \frac{\text{Sum}_{i-1}}{N} + \text{Frame}_i \tag{III.1}$$

$$\text{Pedestal}_i = \frac{\text{Sum}_i}{N} \tag{III.2}$$

This way we only need to store the sum of the last N frames which is $1024 * 1024 * 4 = 4MB$

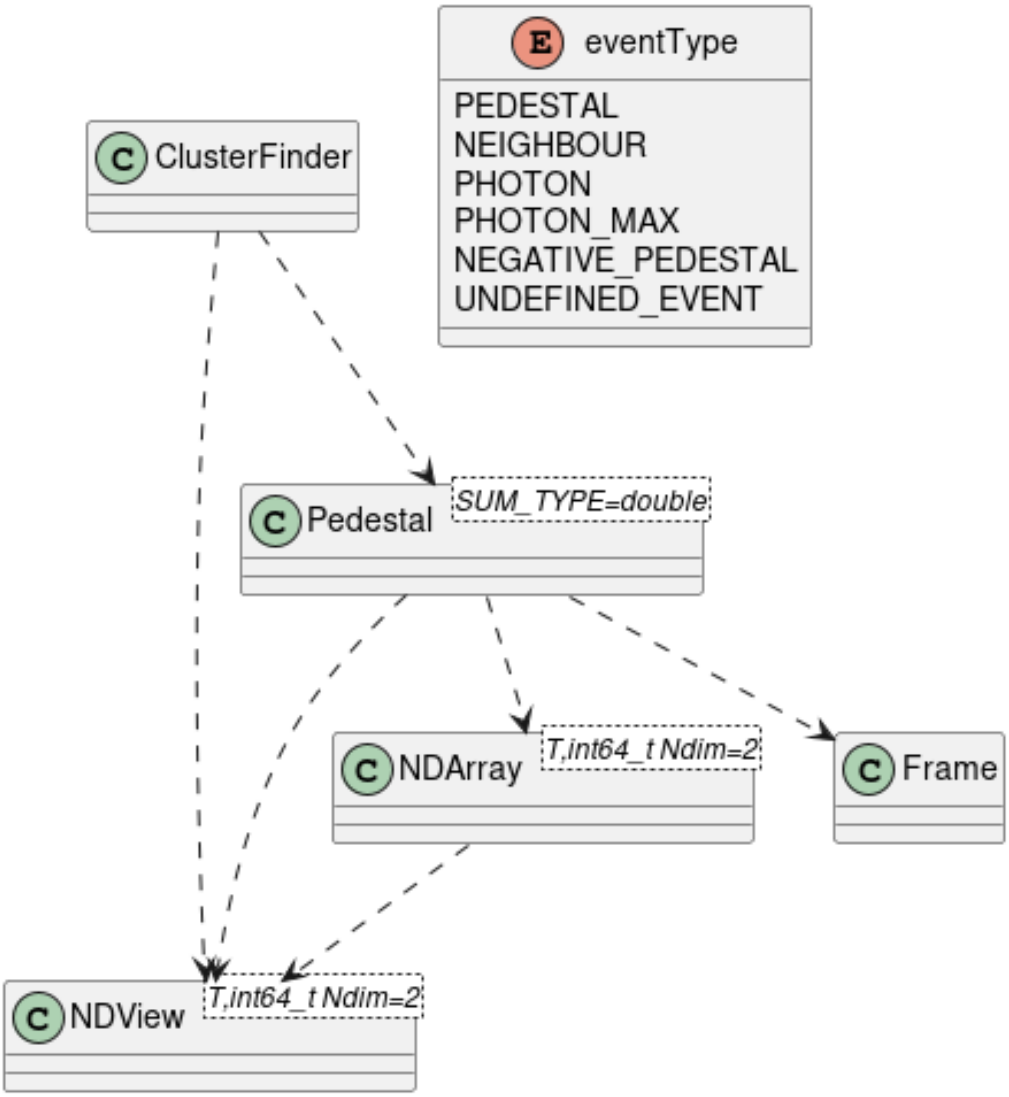


Figure III.6 – Processing module class diagram

of memory. In addition this method is faster, more stable and $O(1)$ in terms of memory and time complexity.

4.2 ClusterFinder class

After substracting the pedestal values from the incoming frame data, the next step is to find the clusters of pixels that are hit by the incoming particles. For photon particles a fixed size cluster of 2x2 or 3x3 is used as a sliding window to find the appropriate particle hits. For electrons, as they usually leave a trail behind, a flood fill algorithm is used to find the clusters of adjacent pixels that are hit by the incoming particles.

5 Python Bindings Implementation

Writing python code is a lot of fun and is very productive. It is a high-level language that is easy to learn and use. It is also very popular in the scientific community. It is used for data analysis, machine learning, artificial intelligence, web development, automation and many more.
However, Python is an interpreted language and it is very dynamic. this makes it slower than compiled languages like C++.

On the other hand, C++ is a compiled language that is very fast and efficient. It is used to write high-performance code that is used in games, operating systems, databases, web servers , financial systems and many more.
It is also very complicated and hard to learn and use. It is strongly typed and has a lot of boilerplate code.

Ideally, we would like to write the core of our library in C++ and then use it in Python. We write high level code in Python benefetting from its simplicity and dynamic features. And we leverage the performance of C++ for the internal library code.
This is where Python bindings come in. Python bindings are used to expose C++ code to Python. There are many ways to create Python bindings such as SWIG, Boost.Python, pybind11, ctypes, CFFI...
In this project, we use pybind11 to create the Python bindings. pybind11 is a lightweight header-only library that exposes C++ code to Python. It is influenced by the Boost.Python library, But it presents itself as bloat free alternative. It is easy to use and provides a lot

of features such as automatic type conversion, exception handling, class inheritance, function overloading, lambda functions and integrates NumPy support.
pybind11 is heavily used in numerical/ML libraries such as scipy, Tensorflow, JAX, Pytorch...

The pybind11 creates an extension module that can be imported in Python. The extension module contains the C++ code that is exposed to Python. After compiling and linking pybind11 produces a dynamic library (dynamic load) that can be used by Python. The dynamic library is loaded at runtime and the C++ code is executed in the Python interpreter.

Listing III.9 – Example of a C++ binding for the Frame class

```
py::class_<Frame>(m, "Frame", py::buffer_protocol())
    .def(py::init<std::byte *, int64_t, int64_t, Dtype>())
    .def(py::init<int64_t, int64_t, Dtype>())
    .def_property_readonly("rows", &Frame::rows)
    .def_property_readonly("cols", &Frame::cols)
    .def_property_readonly("bitdepth", &Frame::bitdepth)
    .def_property_readonly("size", &Frame::bytes)
    .def_property_readonly("data", &Frame::data, py::return_value_policy::
       reference)
    .def_buffer([](Frame &f) -> py::buffer_info {
        Dtype dt = f.dtype();
        return {
            f.data(),                               /* Pointer to buffer */
            static_cast<int64_t>(dt.bytes()),       /* Size of one scalar */
            dt.format_descr(),                      /* Python struct-style format
                descriptor */
            2,                                      /* Number of dimensions */
            {f.rows(), f.cols()},                   /* Buffer dimensions */
            {f.cols() * dt.bytes(), dt.bytes()} /* Strides (in bytes) for
               each index */
        };
});
```

Listing III.9 shows an example of a C++ binding for the Frame class. `.def()` method is used to define a class method. In the example, we defined the constructor with it. `.def_property_readonly()` is used to define a read-only property. `.def_buffer()` is used to define a buffer protocol. The buffer protocol is used to expose the data of the Frame class to Python. This protocol is very useful when working with NumPy arrays as it allows us to convert a Frame object to a NumPy array with or without copying the data.

Listing III.10 – Example of using the Frame class in Python

```
import numpy as np # import numpy
import aare        # import the aare library

frame = aare.Frame(1024, 1024, aare.Dtype.UINT16) # create a frame object
rows = frame.rows # get the number of rows
array = np.array(frame.data, copy=False) # convert the frame object to a
    numpy array
```

Listing III.10 shows an example of using the Frame class in Python. As we can see the Frame class is used as a normal Python class. Furthermore, we can manipulate the data of the Frame object as a NumPy array. This is very useful as it allows us to use the Frame object with other libraries that use NumPy arrays.

Conclusion

In this chapter, we went through the implementation of the project. We discussed the project setup and we dived deep into the details of the implementation. We dived into the design choices that were made in order to juggle between performance, flexibility, and usability.

Part IV

Chapter 4

Chapter IV

Library Applications and Evaluation

Summary

Introduction

In previous chapters, we went through the high level architecture and the implementation of the library. In this chapter, we will explore how the components of the library can be combined to realize different use cases. We will also discuss the parallelization features of the library and how it can be used to improve the performance of the applications. Finally, we will present the results of the performance evaluation of the library.

1 Examples of Library Applications

1.1 Receive frames, Analyze data, Save results

First, we will go through a basic but very common example of using the library. A typical use case of the library is to listen to the slsReceiver for frames, run the cluster finder on the frames as they come in, and store the results in a local file. This example combines the four modules of the library: core, file_io, network_io and processing. The code snippet below IV.1 shows how this can be done in a few lines of code.

Listing IV.1 – Example of a common use case

```
ZmqSocketReceiver receiver("tcp://localhost:5555");
receiver.connect(); // connect to slsReceiver
ClusterFinder clusterFinder(3,3); // 3x3 cluster
ClusterFile clusterFile("clusters.txt","w"); // write to cluster file
Pedestal pedestal(512,512,1000); // pedestal with size 512x512 and averages
    over last 1000 frames

while(true){
    // receive a frame
    ZmqFrame zmq_frame = receiver.receive_zmqframe();
    Frame frame = zmq_frame.frame;
    // find clusters in the frame
    std::vector<DynamicCluster> clusters = clusterFinder.
        find_clusters_without_threshold(frame,pedestal);
    // write clusters to file
    ClusterHeader header = {frame.header, clusters.size()};
    clusterFile.write(header, clusters);
}
```

The library provides a high level interface to the user, abstracting a lot of the implementation details. The scientist can focus on the data analysis part of the code while the library provides reliable, efficient and potent tools to handle the rest.

1.2 File Streaming

Another common use case is to read frames from a file and stream them to one or multiple receivers. This can be useful for testing the performance of the library or for debugging. The file streamer can integrate with other existing systems such as GUI applications or data processing pipelines.

Listing IV.2 – Example of file streaming

```
File file("data_master.json"); // open file for reading
ZmqSocketSender sender("tcp://*:5555"); // create a socket to send frames
sender.bind(); // bind to port 5555
// iterate over all frames in the file
for(int i=0; i<file.total_frames(); i++){
    Frame frame = file.read_frame(); // get frame
    // create a header and fill its metadata
    ZmqHeader header;
    header.frame_number = i;
    // create a zmq frame and send it
    ZmqFrame zmq_frame = {header, frame};
    sender.send_zmqframe(zmq_frame);
}
```

1.3 Load Balancer

As the detector technology evolves, the data rates are increasing. Scaling vertically by increasing the processing power of a single machine is a good solution but it can only go so far. Scaling horizontally by adding more machines to the processing pipeline is a more sustainable solution. As explained in the previous chapter III.5 III.6 III.7 III.8, the library provides a simple way to distribute tasks to multiple workers. The library can be used to create a load balancer that receives frames from a source, and distribute them to multiple data processing pipelines.

1.4 Middleware

A middleware is a software layer that sits between two applications. It has many uses such as for message oriented communication, for cloud services, abstracting hardware details...
In our context the library can be used as a middleware between two systems. For example, the library can sit between the slsReceiver and a GUI application to filter data, apply transformations and more. It can also be used to log data, to limit the rate of data transfer, to merge frames...
It is also possible to implement multiple binaries that use the library and chain them together to create a complex data processing pipeline. The possibilities are endless, and the library's flexibility allows for a wide range of applications.

1.5 Setup Computation Cluster

During the development of the task distribution system, we have seen that the library can be used to offload tasks to multiple workers. A curious idea is to use the library to create a computation cluster. Several on-premise lab computers can be used as workers. The worker would always be waiting for incoming tasks. The ventilator can specify in the Task object the type of processing to be done. Example: ”find clusters”, ”apply threshold”, ”apply mask”... Sinks can also be used to synchronize between workers.
After the setup, several scientists can use the cluster to analyze data in real-time. The cluster can also be used to process data in batch mode.

In this setup, we can have multiple ventilators. Each user will take the role of a ventilator. Tasks will then be fairly queued to the workers and processed in parallel. It is also practical to hide the workers behind a proxy server. The proxy can be implemented with ZeroMQ. The proxy is important so that it can be assigned a static IP address or a domain name. Then workers would only need to discover this proxy and connect to it.

2 Parallelization and Multi-threading

In many cases single threaded applications can become a bottleneck. Running multiple threads can parallelize the work on multiple cores and improve the performance of the application.

2.1 C++ Parallelization

The library is designed to be thread safe and can be used in multi-threaded or multi process applications. The library uses the C++17 standard library for multi-threading. In the C++ interface a MultiThread class is provided. It is a simple wrapper around the std::thread class. The user can create a MultiThread object and pass a lambda functions to it.

Listing IV.3 – Example of using MultiThread class

```
// user defines function to be executed in parallel
void execute(int offset, int step){
    File file("data_master.json");
    // given offset and step read every step-th frame starting from
       offset
    // e.g. if offset=0 and step=4, read frames 0,4,8,12...
    for(int i=offset; i<file.total_frames(); i+=step){
        Frame frame = file.read();
```

```
            // do something with the frame
        }
    }

    // create a vector of functions
    std::vector<std::function<void()>> functions;
    int N_THREADS=4;
    for(int i=0; i<N_THREADS; i++){
        functions.push_back(std::bind(&execute, i, N_THREADS));
    }

    // create a MultiThread object and run the functions in parallel
    MultiThread mt(functions);
    mt.run();
```

Example IV.3 shows how to use the MultiThread class to run a function in parallel. The user defines a function that takes two arguments: an offset and a step. The function reads frames from a file starting from the offset and reads every step-th frame. The user then creates a vector of functions. Each function in the vector reads a different set of frames. The MultiThread object is created with the vector of functions and the run method is called to execute the functions in parallel.

In case we need to synchronize between threads, it is encouraged to avoid C++ mutexes and locks. These concepts are always error prone and significantly increase the complexity of the code. Instead it is preferred to use the ZeroMQ recommendation of using a message passing system between threads. ZeroMQ provides a simple and efficient way to communicate between threads and processes. This alternative approach sacrifices some performance for simplicity and reliability.

> *Code that wants to scale without limit does it like the Internet does, by sending messages and sharing nothing except a common contempt for broken programming models.* [Hin13]

Furthermore, we can achieve parallelization with multiple processes. The same setup that can be used to distribute tasks to mulltiple machines over the network can be used to distribute tasks to multiple processes on the same machine. The library can setup a process that acts as the ventilator and sink. After compilation, we can launch multiple instances of the worker binary.

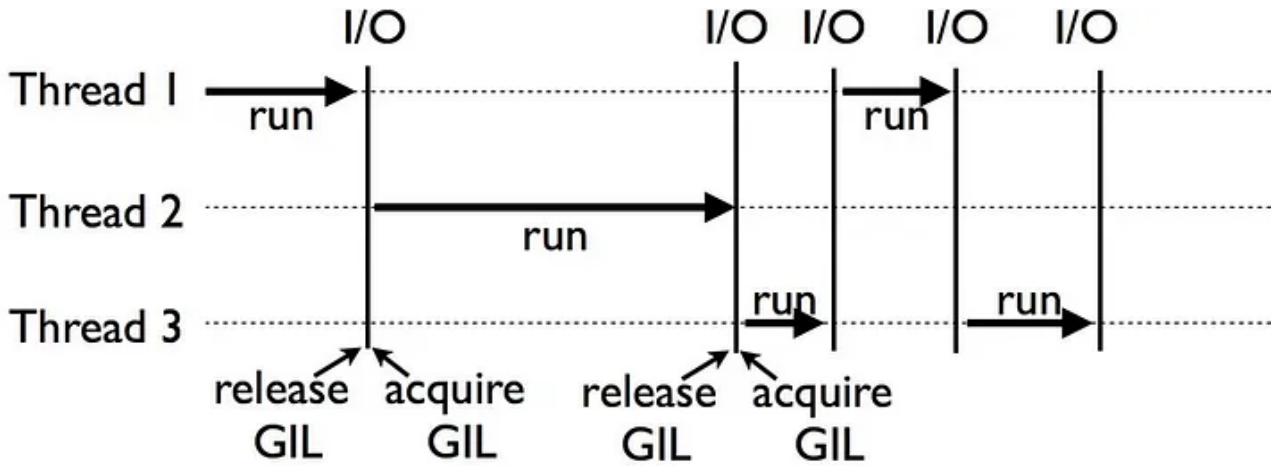


Figure IV.1 – Illustration of thread concurency in Python. The GIL prevents multiple threads from executing Python bytecodes in parallel.

2.2 Python Parallelization

Python first appeared in 1991, it was developed by Guido van Rossum as a scripting language. Python was designed in a time where multi-core processors were not common. The vision of the language was to be simple and easy to use. For this reason, the CPython implementation of Python (or what is commonly refered to as Python) has a Global Interpreter Lock (GIL). The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. Python uses a reference counting garbage collector. The GIL is necessary to protect the reference counting mechanism. Although the GIL was a reasonable design choice at the time, it has become a bottleneck for multi-threaded applications.

The GIL is a controversial topic in the Python community. Some argue that the GIL is a necessary evil, others argue that it is a design flaw. The GIL has been the subject of many debates and discussions. Several attempts have been made to remove the GIL from Python. Although there is an experimental attempt to have the option of disabling it, at the time of writing this report, the GIL is still present in CPython.

The GIL is not a problem for I/O bound applications. Threads can be used to handle I/O operations such as reading from a file, sending a request to a server, waiting for a response... The GIL is released when a thread is waiting for I/O. This allows other threads to execute Python bytecodes concurrently. Figure IV.1 illustrates how the GIL affects thread concurrency in Python.
The GIL, however, is a problem for CPU bound applications. Threads cannot be used to paral-

lelize CPU bound tasks. The GIL prevents multiple threads from executing Python bytecodes at once. This means that a multi-threaded Python application will not be able to take advantage of multiple cores.

A partial solution to the GIL problem is to use multiple processes instead of multiple threads. Each process will have its own Python interpreter and its own GIL. This allows multiple processes to run Python bytecodes in parallel. The multiprocessing module in Python provides a simple way to create multiple processes. However, the overhead of creating multiple processes can be significant.
The benefit of using threads is that threads from the same process share the same memory space. This makes it easy to share objects between threads. Processes, on the other hand, do not share memory space. This complicates the sharing of data and introduces complex concepts in the code such as inter-process communication, shared memory...

As our library is written internally in C++, we can benefit from the multi-threading capabilities of C++ from Python. Python code is not thread safe. However, out library's code is. Knowing this, we can make sure that when calling C++ code from Python, we release the GIL before invoking C++ code and acquire it back after the C++ code has finished executing. This way we can guarantee that Python bytecode is not executed in parallel while C++ code can run in parallel.

As long as we do not alter Python objects in the C++ code, we can safely run C++ code in parallel from Python. This is a powerful feature that can be used to improve the performance of Python applications.
This simple trick allowed us to run threaded code in Python in parallel working around the GIL limitations.
Figure IV.2 illustrates the execution of a multithreaded python code. As we can see from the figure, the GIL is released before invoking the C++ code and acquired back when running python code. Python code is not executed in parallel while C++ code can run in parallel. Effectively, we have bypassed the GIL limitations and achieved parallelization in multithreaded Python.

2.3 Performance Evaluation

In this sub section we will evaluate the different parallelization techniques and compare their performance. As a benchmark, we will consider the common task of reading frames from a

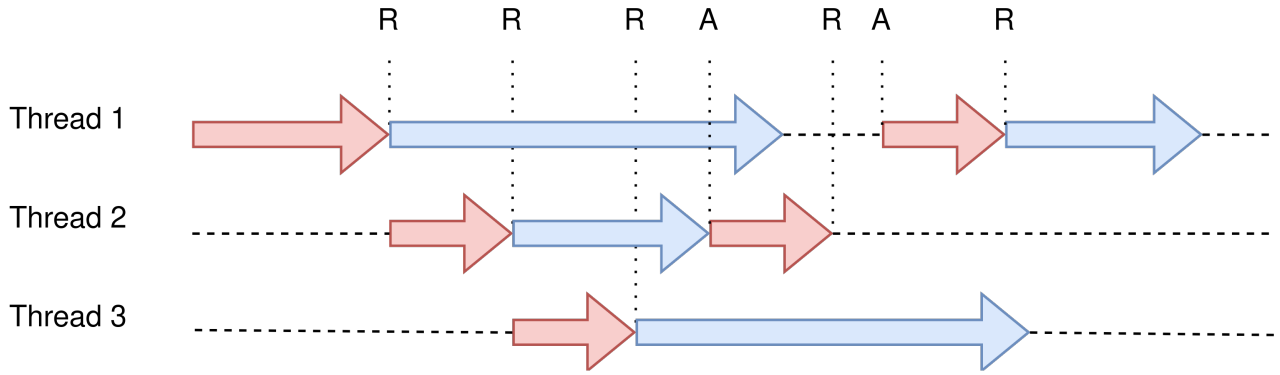


Figure IV.2 – Illustration of running multithreaded Python code with C++ bindings. Red arrows represent Python code, blue arrows represent C++ code. letter R represents releasing the GIL, letter A represents acquiring the GIL.

file, applying pedestal subtraction and finding clusters. This task contains I/O operations for reading frames from a file, and CPU bound operations for pedestal subtraction and cluster finding. We will evaluate the performance of the task using single threaded, multi-threaded and multi-process implementations.

2.3.1 Impact of the GIL

The python interface of the library uses C++ bindings to call the internal C++ code. As discussed in the previous chapter, the GIL prevents the parallelization of Python threads. Python threads run concurrently for IO bound tasks. However, for CPU bound tasks, the GIL prohibits the parallel execution of Python bytecodes.

On the other hand, C++ code is thread safe and it is not affected by the GIL. For this reason, when calling C++ code from Python, we can release the GIL, run C++ code and acquire it back once the C++ code has finished executing. This way from Python's perspective the C++ code will treated the same as IO bound tasks and Python bytecodes will not be executed in parallel.

To evaluate the impact of the GIL on our library we created two versions of the library. The first version releases the GIL before invoking the C++ code and the second one does not. We then ran the same task on both versions and compared the performance.

Table IV.1 shows the impact of the GIL on the performance of the library. The table compares

Thread Count	With GIL		Without GIL	
	Time (s)	CPU Usage	Time (s)	CPU Usage
1	107	97%	105	96%
2	123	96%	71	189%
3	135	96%	65	265%
4	140	96%	58	352%
8	130	96%	58	365%

Tableau IV.1 – Impact of the GIL on the performance of the library. Comparing the time taken to run a task and the CPU usage on a 4-core machine.

the time taken to run a task and the CPU usage for different thread counts. The task was run on a 4-core machine. CPU usage is calculated as the sum of the CPU usage of all cores. As we can see from the table, the time taken to run the task is significantly higher when the GIL is present. The CPU usage is also lower when the GIL is present. This is because the GIL prevents the parallel execution of C++ code. The as we use multiple threads we can see that the performance degrades as the number of threads increases. This is due to the cost of context switching between threads.

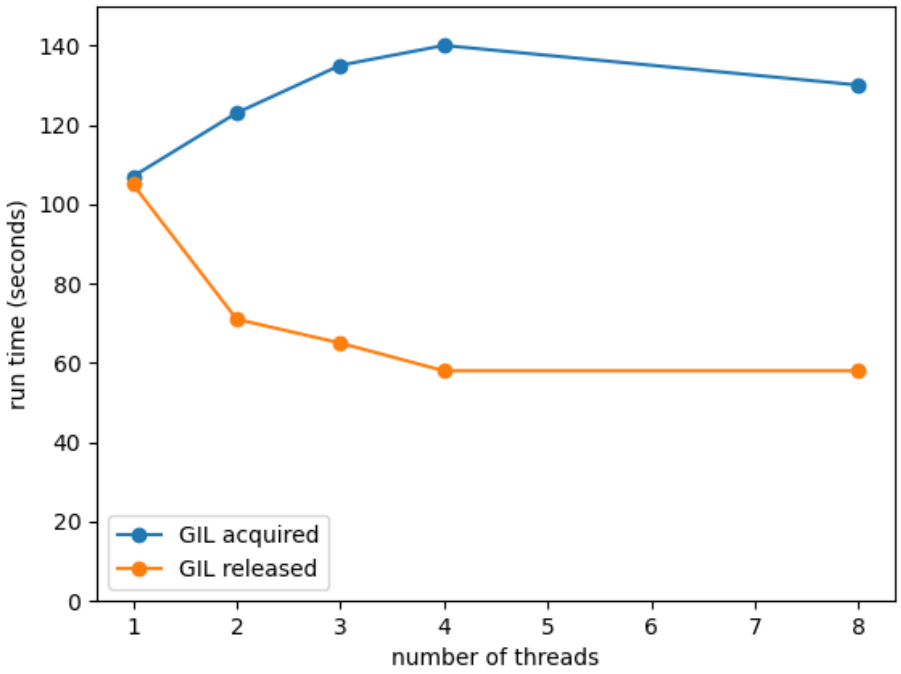
In comparison when the GIL is not present, the time taken to run the task is significantly lower. CPU Usage quadrupled from 96% to 365%. The multiple threads are running in parallel and the performance improves as the number of threads increases. Figure IV.3 plots the table data. As we can see from the plot, the performance of the library is significantly improved when the GIL is released before calling C++ code.

Naturally, the performance of the C++ library is not affected by the GIL as it only impacts Python code.

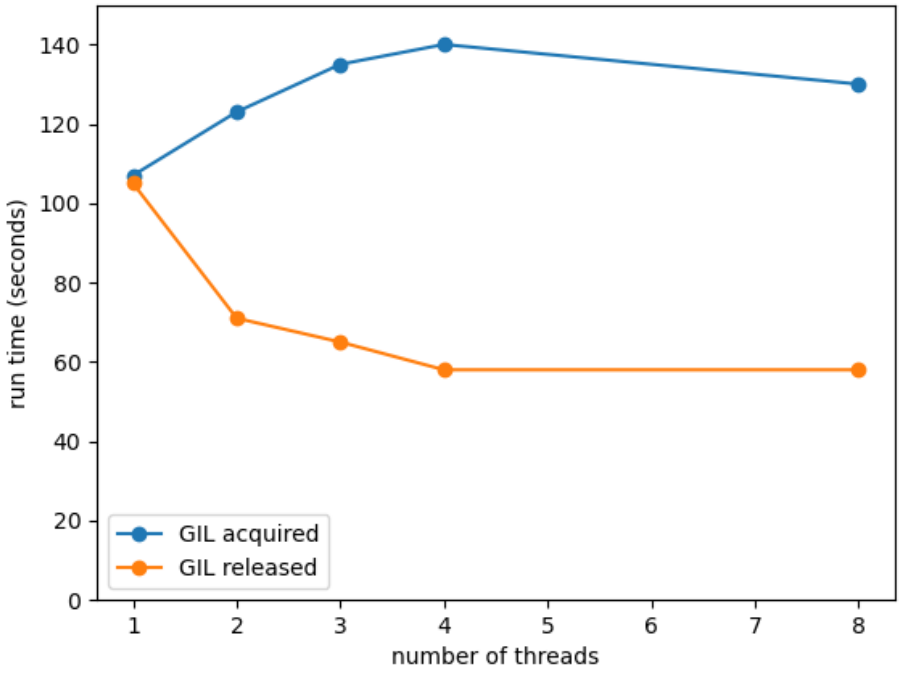
2.3.2 Comparison of Different Parallelization Techniques

The next step is to compare the performance of different parallelization techniques. Four techniques are implemented: multi-threaded Python, multi-threaded C++, multi-process C++ with ZeroMQ, multi-threaded C++ with Producer and Consumer Queues.

The experiments were done on a cloud virtual machine deployed on Azure Cloud. The machine used has a 4-core CPU and 16 GB of RAM (D4s_v3). The benchmark as previously defined is to read 8000 frames from a file, apply pedestal subtraction and find clusters.



(a) Run Time



(b) CPU Usage

Figure IV.3 – Plot showing the impact of releasing the GIL in the python bindings

Examples of the multi-threaded C++ and Python implementations are shown in listings IV.4 and IV.5. The same idea is implemented exactly in python and in C++. In these examples a list of functions is created. Each function reads a different set of frames from the file independently from the others. These functions will then be run in threads in parallel.

The multi-threaded C++ with Producer and Consumer Queues implementation is similar to the multi-threaded C++ implementation. The difference is that instead of running the functions in threads directly, a thread will push frames to a queue and other threads will pop frames from the queue and process them. This is a common pattern used in multi-threaded applications. In our comparison it serves as a reference point to compare the performance of the other implementations.

The multi-process C++ with ZeroMQ implementation is based on the task distribution model implemented in the library. The ventilator, worker, sink model is designed to work over the network but it can also be used to distribute tasks to multiple processes on the same machine. The ventilator will read frames from the file and send them to the workers over ZeroMQ. All communications will be local to the host machine and will not go over the network.

Listing IV.4 – Multi-threaded C++ example

```
const int N_FRAMES = 8000;
std::filesystem::path fname;

std::function<void()> thread_function_factory(int offset, int step) {
    // return a function that will be run in parallel
    // the function has a closure over the offset and step
    // in case of 4 threads, step will be equal to 4 and offset will be
       0,1,2,3
    // the function reads frames from the file starting from the offset and
       reads every step-th frame
    // one function will read frames 0,4,8,12... another will read frames
       1,5,9,13...
    return [offset, step]() {
        Pedestal<double> pedestal(400, 400, 1000); // create pedestal object
        File file(fname, "r"); // open file
        ClusterFinder cf(3, 3, 5, 0); // inialize cluster finder
        for (int i = offset; i < N_FRAMES; i += step) {
            // read frame and run cluster finder algorithm
            auto frame = file.iread(i);
```

Thread/Process Count	MT C++	MT Python	MT C++ PCQ	MP C++ ZMQ
1	74	98	90	143
2	37	51	46	72
3	37	49	46	70
4	35	50	47	67
8	36	47	43	68

Tableau IV.2 – Comparison of different parallelization techniques. Time taken to run the benchmark in seconds. MT C++: Multi-threaded C++, MT Python: Multi-threaded Python, MT C++ PCQ: Multi-threaded C++ with Producer and Consumer Queues, MP C++ ZMQ: Multi-process C++ with ZeroMQ.

```
            auto clusters = cf.find_clusters_without_threshold(frame.view<
                uint16_t>(), ped, false);
        }
    };
}
```

Listing IV.5 – Multi-threaded Python example

```
# function that will be run in parallel
def executor(offset,n_threads):
    file = File(file_path)
    pedestal = Pedestal(400,400,1000)
    cf = ClusterFinder(3,3,5.0,0)
    for i in range(offset,N_FRAMES,n_threads):
        frame = file.iread(i)
        clusters=cf.find_clusters_without_threshold(frame,pedestal,False)



thread_arr = []
for i in range(N_THREADS):
    # create and run threads
    thread_arr.append(Thread(target=executor,args=(i,N_THREADS)))
    thread_arr[i].start()
```

Table IV.2 and figure IV.4 show the time taken to run the benchmark for different parallelization techniques. The multithreaded C++ example is the fastest. It took 35 seconds to run the test on 4 threads. Naturally, the multi-process ZeroMQ example is the slowest. It is always twice slower than the multi-threaded C++ example.

Surprisingly, python performed close results to C++. The multi-threaded Python example

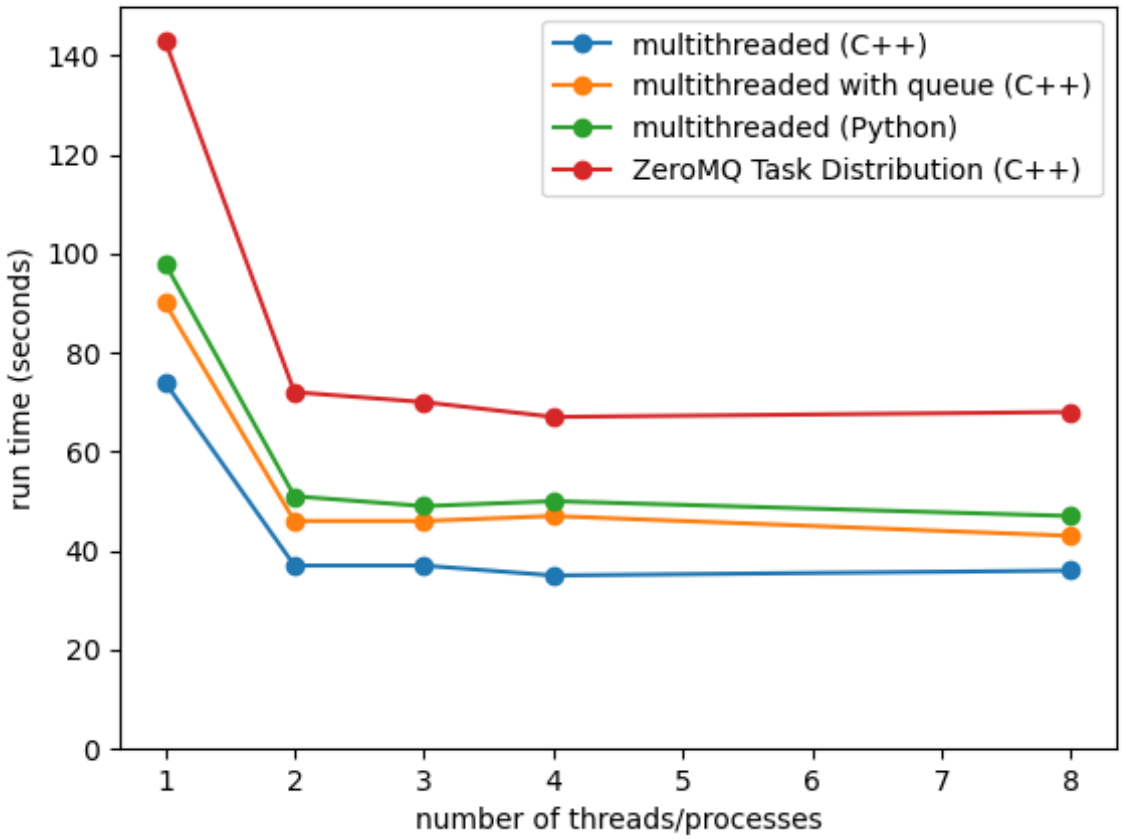


Figure IV.4 – Comparison of different parallelization techniques. Time taken to run the benchmark in seconds.

took 47 seconds to run on 8 threads. While the producer consumer C++ example took 43 seconds. Compared to the multithreaded C++ example, it is in general 35% slower. This is a good result for Python! This is orders of magnitude faster than code written in pure Python.

When designing the library, it was first thought that the same task distribution model could be used as a parallelization framework locally. The results of this evaluation show that the model is not as efficient as a multi-threaded implementation in C++. As much as it is convenient if we could reuse the same code to scale it horizontally and vertically it is not practical. On the other hand multi-threaded code proved itself as the most efficient way to parallelize and it comes with the least overhead.

Conclusion

In this chapter we explored different use cases of the library. We demonstrated the flexibility and power of the library by combining its components to realize different applications. We also discussed the parallelization features and benchmarked its performance with various parallelization techniques. The results show that the library is efficient and can be used to handle complex tasks in simple ways and with high performance.

Conclusion and Perspectives

Bibliography

[BA22] William Bugden and Ayman Alahmar. “Rust: The programming language for safety and performance”. In: *arXiv preprint arXiv:2206.05503* (2022).

[Bög02] M Böge. “First operation of the swiss light source”. In: *Proc. EPAC*. Vol. 2. 2002.

[But+24] Tim A Butcher et al. “Ptychographic nanoscale imaging of the magnetoelectric coupling in freestanding BiFeO3”. In: *Advanced Materials* (2024), p. 2311157.

[CR14] James O Coplien and Trygve Reenskaug. “The dci paradigm: Taking object orientation into the architecture world”. In: *Agile Software Architecture*. Elsevier, 2014, pp. 25–62.

[Dul+24] Christian Dullin et al. “In vivo low-dose phase-contrast CT for quantification of functional and anatomical alterations in lungs of an experimental allergic airway disease mouse model”. In: *Frontiers in medicine* 11 (2024), p. 1338846.

[Fou22] Python Software Foundation. *Buffer Protocol*. 2022. URL: https://docs.python.org/3/c-api/buffer.html.

[FP09] Steve Freeman and Nat Pryce. *Growing object-oriented software, guided by tests*. Pearson Education, 2009.

[Gam+04] Louay Gammo et al. “Comparing and evaluating epoll, select, and poll event mechanisms”. In: *Linux Symposium*. Vol. 1. 2004.

[Har+20] Charles R Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[Hin13] Pieter Hintjens. *ZeroMQ: messaging for many applications*. ” O’Reilly Media, Inc.”, 2013.

[JG10] Muris Lage Junior and Moacir Godinho Filho. “Variations of the kanban system: Literature review and classification”. In: *International journal of production economics* 125.1 (2010), pp. 13–21.

[JRM17] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017.

[Lem+23] Tali Lemcoff et al. “Brilliant whiteness in shrimp from ultra-thin layers of birefringent nanospheres”. In: *Nature Photonics* 17.6 (2023), pp. 485–493.

[Leo+23] Filip Leonarski et al. “Kilohertz serial crystallography with the JUNGFRAU detector at a fourth-generation synchrotron source”. In: *IUCrJ* 10.6 (2023).

[Lis87] Barbara Liskov. “Keynote address-data abstraction and hierarchy”. In: *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*. 1987, pp. 17–34.

[Lv2222] Q. Z. Lv et al. “High-Brilliance Ultranarrow-Band X Rays via Electron Radiation in Colliding Laser Pulses”. In: *Phys. Rev. Lett.* 128 (2 Jan. 2022), p. 024801. DOI: 10.1103/PhysRevLett.128.024801. URL: https://link.aps.org/doi/10.1103/PhysRevLett.128.024801.

[Mar00] Robert C Martin. “Design principles and design patterns”. In: *Object Mentor* 1.34 (2000), p. 597.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[Mar15] Nikhil Marathe. “An Introduction to libuv”. In: *Nikhilm. github. io* 2 (2015).

[MarBlog12] Robert C. Martin. “The Clean Architecture”. In: (2012). URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[Martin17] Robert C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. 1st. USA: Prentice Hall Press, 2017. ISBN: 0134494164.

[Mey97] Bertrand Meyer. *Object-oriented software construction (2nd ed.)* USA: Prentice-Hall, Inc., 1997. ISBN: 0136291554.

[Palermo08] Jeffrey Palermo. “The Onion Architecture”. In: (2008). URL: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/.

[Pik18] Colby Pike. *Pitchfork is a Set of C++ Project Conventions*. 2018.

[Pom+09] Daniel A Pomeranz Krummel et al. “Crystal structure of human spliceosomal U1 snRNP at 5.5 Å resolution”. In: *Nature* 458.7237 (2009), pp. 475–480.

[Web] PSI Website. *About Swiss Light Source SLS*. URL: https://www.psi.ch/en/sls/about-sls.