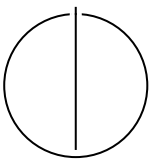
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design, Implementation and Test of a Python API for the sys-sage Library

Jonathan Wieser



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

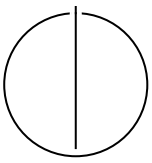
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design, Implementation and Test of a Python API for the sys-sage Library

Design, Implementierung und Test einer Python API für die sys-sage Library

Author:	Jonathan Wieser
Examiner:	Prof. Dr. rer. nat. Martin Schulz
Supervisor:	Stepan Vanecek
Submission Date:	14.04.2025



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 14.04.2025 Jonathan Wieser

Acknowledgments

Abstract

High-Performance Computing (HPC) continues to be a prominent field, characterized by increasingly intricate topologies and the growing integration of accelerators such as GPUs. In such complex systems, analyzing resource allocation and system states is crucial, encompassing both static infrastructure and dynamic runtime behavior to identify bottlenecks and adapt to system changes. While existing tools address static and dynamic analysis separately, Sys-Sage uniquely combines these capabilities, offering a broad spectrum of functions to examine components and their interconnections. Its architecture allows users to tailor and extend functionalities to specific use cases, albeit requiring C++ proficiency.

To enhance usability and facilitate rapid prototyping for a wider audience, particularly those seeking an intuitive way to interact with Sys-Sage without extensive compilation, this work focuses on providing a Python interface. Python, a familiar language with a relatively gentle learning curve, is well-suited for prototyping and extending existing libraries. We analyze and compare the usability and performance characteristics of different technologies for achieving this Python binding, ultimately selecting and implementing a suitable solution. Furthermore, we introduce a new feature to the core Sys-Sage library in C++: XML import functionality. Comprehensive benchmarks are conducted to quantify the performance difference between the resulting Python interface and the native C++ implementation. The thesis concludes with a discussion of future work, including the integration of modified destructors within the Sys-Sage library to improve memory management and simplify the user experience.

Contents

1 Introduction

1.1 Introduction to sys-sage Library

High Performance Computing (HPC) systems have undergone significant transformations, becoming increasingly complex. This complexity arises from the integration of various components, particularly the growing reliance on multi-core processors and Graphics Processing Unit (GPU)s. As modern architectures evolve, understanding system behavior becomes more challenging, necessitating more sophisticated analysis tools. Several approaches exist for analyzing system architectures, with one of the most commonly used tools being `hwloc`. This tool facilitates the construction of hardware topologies, mapping essential building blocks such as cores, caches, and Non-Uniform Memory Access (NUMA) nodes. While `hwloc` provides fundamental insights into system topology and application scheduling, it is limited to Central Processing Unit (CPU)-based static analysis. However, contemporary architectures introduce dynamic resource allocation and isolation, rendering static analysis insufficient for comprehensive system evaluation. Consequently, dynamic analysis becomes increasingly valuable in modern computing environments. Given the growing dynamism of HPC systems, a unified tool that consolidates the advantages of existing solutions while incorporating real-time hardware and resource utilization changes would be highly beneficial. [23] [17]

The `sys-sage` library addresses this need by offering a unified Application Programming Interface (API) for accessing the state of modern architectures. As an open-source solution, `sys-sage` is highly adaptable and supports the integration of diverse datasets, such as GPU benchmarks and power efficiency metrics, enabling a comprehensive understanding of system performance. It constructs an internal representation of hardware topology alongside data paths, allowing for both export and, with this work, import capabilities to facilitate data storage and retrieval. [23] [11]

1.2 Scope of This Work

Currently, `sys-sage` provides support exclusively for C++, limiting its accessibility to a broader audience. Meanwhile, Python has emerged as the most widely used

programming language, surpassing JavaScript with **16.925%** of all GitHub projects developed in python. Given its widespread adoption, introducing a Python API for `sys-sage` would significantly enhance usability, enabling a larger community to leverage its capabilities. [5]

The objective of this work is to develop a Python interface for `sys-sage` while preserving the performance advantages of the underlying C++ implementation. The general idea is to expose essential C++ library functions to Python, allowing users to compile the code once and execute scripts efficiently with the compiled runtime.

In the next chapter, in Chapter 2, we provide an overview of the `sys-sage` library and its components, followed by a presentation of the key differences between C++ and Python. In Chapter 3, we compare different technologies, one of which is used for the Implementation, described in Chapter 4. After that, we present the Evaluation of the solution in Chapter 5. Since this work's contribution is not finalizing the `sys-sage` library, we provide a description of the future work in Chapter 7.

2 Terms and Definitions

2.1 sys-sage Specific Terms

To understand how sys-sage operates, it is essential to define its core concepts. sys-sage is designed with a modular structure, allowing for high adaptability through various compilation options that enable different functionalities. The system is primarily composed of components, data paths and their attributes, along with parsers and external interfaces, each serving a distinct role in modeling and analyzing complex hardware architectures.

2.1.1 Components

At the heart of sys-sage lies the concept of components, which represent various elements of a high-performance computing (HPC) system. The most fundamental component is simply referred to as Component, a generic type that serves as the base class for all other component types. Every component inherits essential attributes such as name, id, and type, ensuring consistency across the system. [22, see Component]

A key component type is Topology, which defines the root of an HPC system. However, sys-sage allows any component to act as the root, offering flexibility in system representation. The Node component represents a computing node within an HPC system and provides specific functions such as RefreshCpuCoreFrequency(), if Proc_Cpu is enabled. This functions dynamically updates CPU frequencies. If the Intel_Pqos extension is enabled, an additional function, UpdateL3CATCoreCOS(), becomes available, offering more control over cache allocation. [22, see Topology, Node]

Memory is another crucial component in the system, represented by the Memory class. This component models various types of memory, with attributes such as size, which can be adjusted, and volatility, which remains static. For systems incorporating NVIDIA MIG technology, the memory component can retrieve the MIGSize of a GPU memory partition. [22, see Memory]

Similarly, the Chip component represents any type of processing unit, including CPUs, GPUs, and accelerators. It includes attributes such as vendor, model, and chip-type. When NVIDIA MIG is enabled, additional functionalities such as UpdateMIGSettings(),

GetMIGNumSMs(), and GetMIGNumCores() provide extended configurability. [22, see Chip]

To allow for hierarchical organization, sys-sage introduces the Subdivision component, which enables user-defined groupings within the system. A specialized subclass of Subdivision is NUMA (Non-Uniform Memory Access), which models memory locality and allows for adjustable memory sizes. [22, see Subdivision, NUMA]

Another critical component is the Cache, which represents cache memory within a system. It allows users to specify cache level, associativity ways, and cache-line size, determining how memory addresses map to cache lines. If NVIDIA MIG is enabled, the function GetMIGSize() becomes available, providing additional insights into GPU memory partitioning. [22, see Cache]

At the processing level, the Core component models individual cores in CPUs or GPU streaming multiprocessors, with frequency attributes that can be both retrieved and modified. In addition, the Thread component represents hardware threads rather than software threads. While it also has a frequency attribute, this can only be read, not modified. If the Intel_Pqos extension is enabled, the function GetCATAwareL3Size() provides additional cache-related information. [22, see Core, Thread]

Ther are not only child-parent relationships. Sys-sage also supports datapaths to connect arbitrary components.

2.1.2 Datapaths

In addition to components, sys-sage models interconnections within an HPC system using datapaths. A datapath defines a link between two components, with each path having a well-defined source and target. While each component can have multiple data paths, individual data paths are unidirectional by default. However, bidirectional data paths can also be modeled, in which case they are stored twice — once as source and once as target. [22, see Data Path]

Two primary attributes define data paths: latency and bandwidth, which are included by default. However, additional attributes can be incorporated depending on the specific use case, including parameters related to data transfer efficiency, system performance, and power consumption. These attributes allow data paths to accurately represent dynamic system behavior and adapt to changing configurations.[22, see Data Path]

Datapaths share the same mechanism for additional attributes as Components, where the user has the option to add custom attributes.

2.1.3 Attributes

Both `Component` and `Datapath` classes contain a set of predefined attributes such as `id` and `name` or `orientation` and `bandwidth` respectively. However, `sys-sage` also allows users to define custom attributes through a flexible key-value mapping system. This feature enables components to store additional metadata, such as frequency history, while data paths can incorporate attributes related to power consumption or other performance characteristics.[22, see Component, Datapath]

Now that we showed the essential parts to build a system topology, we will take a closer look at parsers and external interfaces, that help to build and feed the internal representation with data.

2.1.4 Parsers

To facilitate system analysis, `sys-sage` provides various parsers that allow the integration of external data sources. Supported parsers include `Caps_Numa_Benchmark`, `cccbench`, `hwloc`, and `mt4g`. These parsers enable the system to read relevant data and construct an internal representation of the hardware topology, making `sys-sage` a powerful tool for performance analysis and resource management.[22, see Data Parsers Documentation]

Parsers play a important role in static analysis, but to make `sys-sage` dynamic, the user can also integrate external interfaces.

2.1.5 External Interfaces

As menitioned in the Components section, some functions depend on external interfaces. These interfaces include `Intel_Pqos`, `NVIDIA_MIG`, and `Proc_Cpuinfo`. They allow `sys-sage` to dynamically collect and analyze hardware performance data. However these interfaces are only available for systems, that have access to these technologies.[22]

We now have a overview of `sys-sage` and what role each of the main parts play. In order to grasp the potential challenges for binding the `sys-sage` library to Python, we will discuss the key differences between C++ and Python.

2.2 Differences Between C++ and Python

While `sys-sage` is implemented in C++, Python has emerged as one of the most popular programming languages, surpassing JavaScript with 16.925% of all GitHub projects being coded in Python. Given its popularity, integrating Python support into `sys-sage` offers significant advantages in terms of usability and accessibility. [5]

One of the most notable differences between C++ and Python is performance. C++ is a compiled language, meaning that code is translated directly into machine instructions before execution. In contrast, Python relies on an interpreter, which translates code at runtime. This additional translation step significantly slows down execution speed, with C++ often performing up to 100 times faster than Python. While there are approaches to mitigate Python's performance limitations — such as using the `Numba` compiler[12] — these methods impose restrictions on the language's features and usability.[15]

Another key distinction lies in syntax and readability. Python has gained widespread adoption due to its simple, human-readable syntax, which does not require type declarations, pointers, or other low-level constructs. This makes Python easier to learn and use compared to C++.

Memory management also differs significantly between the two languages. C++ requires manual memory allocation and deallocation, whereas Python employs automatic garbage collection. In `CPython`, objects maintain a reference count, and memory is deallocated once the reference count reaches zero. However, reference counting alone cannot handle cyclic dependencies, which is why Python includes a garbage collector to resolve such issues. While garbage collection simplifies memory management, it also introduces performance overhead. In contrast, C++ provides more control over memory but requires developers to handle allocation carefully to prevent memory leaks.[9]

Another crucial difference is in multithreading and multiprocessing. C++ is well-suited for multithreaded applications and can fully utilize all available processor cores. Python, however, is limited by the Global Interpreter Lock (GIL), which allows only one thread to execute at a time. While this does not pose a problem for **I/O!** (**I/O!**)-intensive applications, it severely impacts CPU-bound tasks. To overcome this limitation, Python developers must use the `multiprocessing` library, which enables parallel execution across multiple processors.[2]

3 Approach

3.1 Requirement Analysis and Technology Comparison

3.1.1 Requirement Analysis

To determine the most suitable solution, a comprehensive requirement analysis is essential. This analysis will also serve to evaluate the effectiveness of the chosen approach and identify any trade-offs made during implementation.

Library Complexity

The sys-sage library, as outlined in the terminology section, exhibits significant complexity, encompassing a multitude of functions. Many of these functions are designed to modify object states and retrieve information about components or the overall topology. The mapping of basic C++ data types to their Python counterparts should be straightforward. Components and Datapaths feature attribute maps, enabling the addition of custom attributes. It is crucial that these attribute maps are accessible from Python, allowing users to define and manipulate custom attributes.

The library also manages horizontal and vertical connections through component parent-child relationships and datapath source-target relationships. The Python implementation must handle all objects with precision, mirroring the execution behavior of the C++ functions. sys-sage employs inheritance, with classes like Topology inheriting from the generic Component class. While Python supports inheritance, the integration of inheritance and polymorphism requires careful consideration to avoid potential workarounds. This aspect will be addressed in this work, particularly concerning the generalization of the Datapaths class to accommodate future specific Datapaths types.

sys-sage supports up to three external interfaces: intel_pqos, nvidia_mig, and proc_cpuinfo. The Python bindings should allow users to utilize these interfaces when available and desired. Additionally, sys-sage provides XML import and export functionality, enabling users to customize attribute import and export processes.

Performance

The most computationally intensive tasks involve searching and updating components within the topology. The time required for these operations increases with the size of the topology. To minimize overhead, the Python bindings should primarily rely on the existing C++ functions, avoiding reimplementation in Python.

Library Dependency

The chosen approach should minimize external dependencies to streamline the installation process and enhance user experience. Excessive dependencies can lead to installation complexities, conflicts, and maintenance challenges.

Error Handling and Documentation

While performance considerations might lead to reduced emphasis on error handling, comprehensive documentation is vital. The documentation should clearly delineate the capabilities and limitations of the Python bindings.

Future Support

Long-term support and maintainability are crucial. The implementation should be easily understandable and modifiable, even by developers unfamiliar with the project.

Build and Installation

Users should have the option to install the library with or without Python accessibility.

User Experience

Python programmers, who may not be accustomed to the intricacies of C++ programming, require an interface that is intuitive and robust. The bindings must prevent errors arising from misuse, such as segmentation faults, by providing clear error messages and idiot-proof functions.

Now that we have our requirements defined we can take a look at the different technologies to reach set goals.

3.1.2 Technology Evaluation for Python Integration

Several technologies were considered for their potential to facilitate the integration of our C++ library with Python. Each option presents a unique set of features and

trade-offs concerning performance, ease of implementation, and compatibility with modern C++ constructs. Furthermore we provided some sketches that show which steps are generally involved to use the technologies in a project. These illustrations should only give a rough idea behind the application of the different options and their complexity.

ctypes: Direct Library Interaction

ctypes enables Python to directly interface with dynamic link libraries, providing access to C-compatible data types and functions. While straightforward for basic C interactions, it necessitates manual type conversions and lacks native support for C++ object-oriented features. This can complicate the wrapping of libraries with complex C++ structures and inheritance. [8]

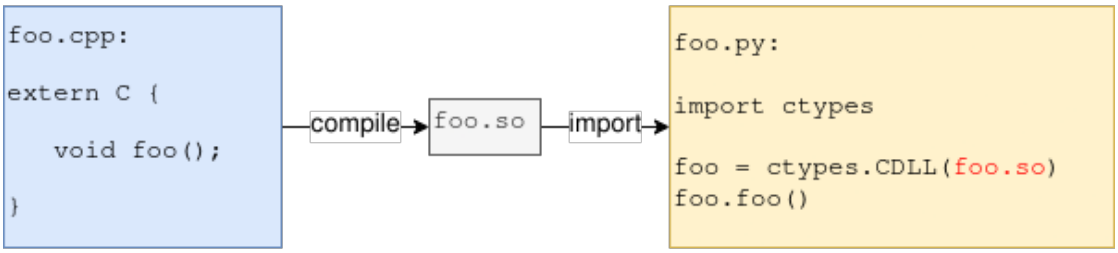


Figure 3.1: ctypes Schema

Cython: Compiled Python Extensions

Cython allows for the compilation of Python-like code into optimized C, potentially yielding performance improvements. It is suitable for creating C extensions and bridging Python with external C libraries. However, its effectiveness in wrapping existing C++ libraries may vary, as it may not fully leverage the library's pre-existing design. [4]

pybind11: Modern C++ Bindings

pybind11 is a header-only library designed to simplify the creation of Python bindings for C++ code. It leverages compile-time introspection to minimize boilerplate and offers robust support for modern C++ features, including templates and Standard Template Library (STL) data structures. Its design prioritizes ease of use and maintainability. [14]

Figure 3.2: Cython Schema

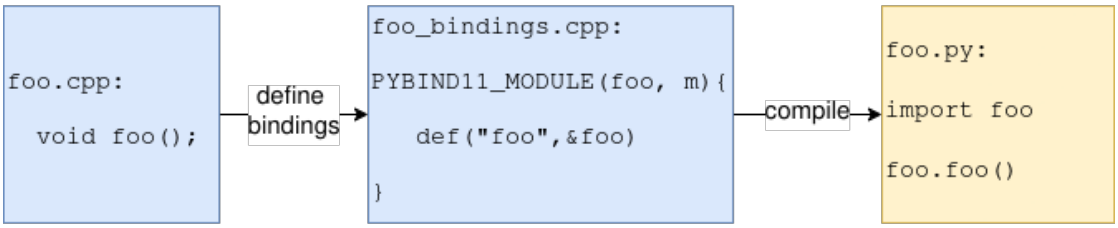


Figure 3.3: pybind11 Schema

Py-C-API: Native Python Extensions

The Python/C API provides a direct mechanism for extending the Python interpreter with C or C++ modules. While offering high performance, it demands meticulous memory management and can result in less readable code. It is generally better suited for creating custom extensions than for wrapping existing libraries. [11]

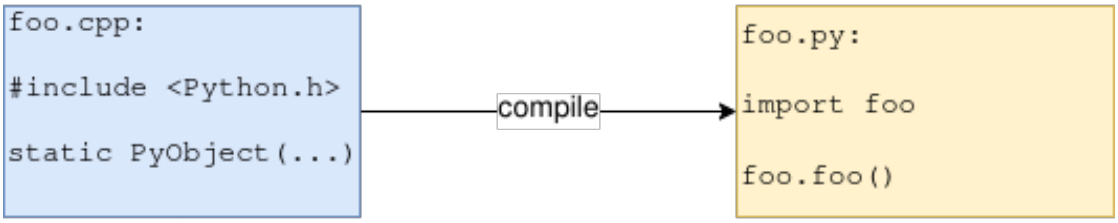


Figure 3.4: Python-C-API Schema

SWIG: Multi-Language Interface Generation

SWIG is a tool capable of generating bindings for numerous programming languages. While it supports a broad range of C++ features, it introduces its own interface definition

language, which may present a learning curve. Additionally, the necessity for manual type mappings can increase development effort. [20]

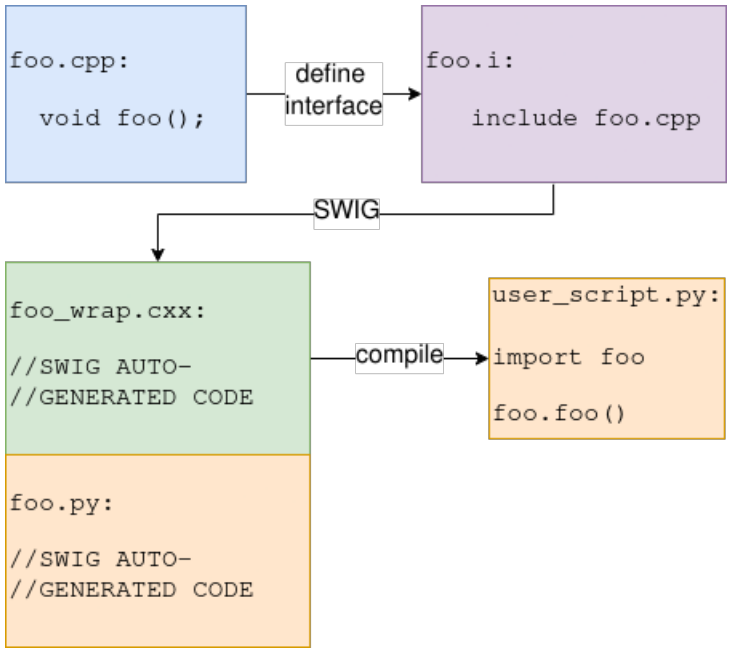


Figure 3.5: SWIG Schema

Alternative Binding Tools

Other tools, such as Boost.Python and Cffi, offer alternative approaches. Boost.Python, while powerful, introduces complex installation process. Cffi, primarily designed for C, may not fully support the complexities of modern C++ libraries, particularly those utilizing modern C++ features. [1] [18]

As we can see there are plenty of technologies to integrate C++ in Python, using different approaches. Checking the performance of these tools is our next step to narrow down our choice.

3.1.3 Performance Comparison

Performance is a critical consideration, especially for tasks involving searching and updating components in large topologies. The overhead associated with each technology, including function calls and type casting, must be evaluated. In the following benchmarks we repeat the task a million times and measure the time it takes.

Initial tests involving file I/O operations indicate that Py-C-API offers the best performance, followed by pybind11 and SWIG. Python and Cython exhibit significantly lower performance due to the interpreter overhead.

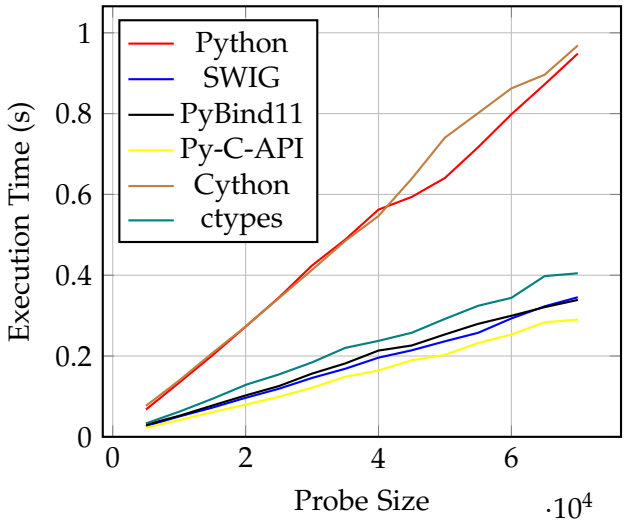


Figure 3.6: Execution time comparison for different Python interoperability methods.

The Python-C-API might be the optimal solution regarding performance, however, the documentation advises to better use a third party tool to simplify the process. [11]

Therefore we only compare the performance of pybind11 and SWIG. To do this we create a datastructure that mimics sys-sage's internal representation in a very basic way. We then measure the time to get an object from a hierarchy of objects.

The results show that SWIG is generally faster, except when using smart pointers. However, with our defined requirements, that not only involve performance we decide to go with pybind11.

3.1.4 Technology Decision: pybind11 for Optimal Integration

The selection of pybind11 as the primary technology for creating Python bindings for the sys-sage library stems from a comprehensive evaluation of various options, balancing performance, maintainability, and ease of use. While performance benchmarks indicated that SWIG might offer a slight speed advantage in certain scenarios, particularly with standard data structures, the crucial consideration of developer experience and long-term maintainability tipped the scales in favor of pybind11.

A significant drawback of SWIG is its reliance on a unique interface definition language. This necessitates that developers learn a new syntax, distinct from both C++

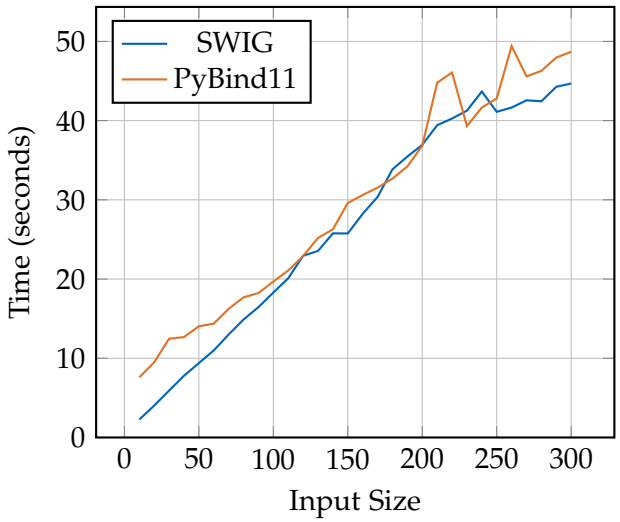


Figure 3.7: Comparison of execution time between SWIG and PyBind11.

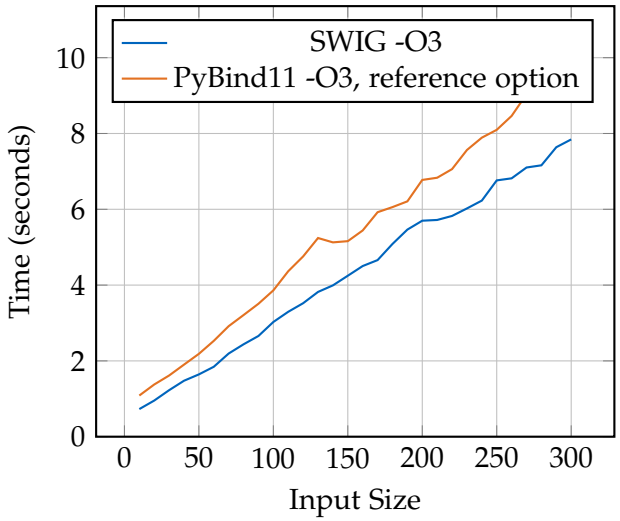


Figure 3.8: A plot comparing the performance of SWIG and PyBind11 with optimization flag -O3.

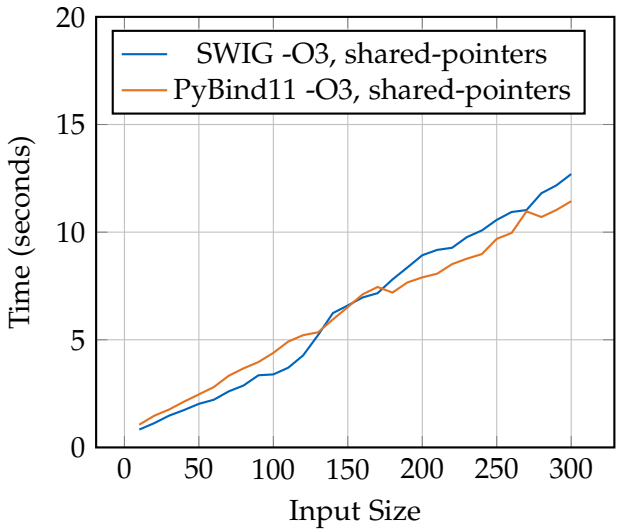


Figure 3.9: A plot comparing the performance of SWIG and PyBind11 with optimization flag -O3 and shared-pointers.

and Python, which can impede development speed and increase the learning curve for new contributors. In contrast, pybind11, being a header-only library embedded within C++, offers a more intuitive and familiar development environment for C++ programmers. This direct integration with C++ simplifies the process of creating and maintaining bindings, making it easier for future developers to understand and extend the project.

This decision aligns with findings from independent benchmarks, such as those presented in the Chapter 6, which highlight pybind11's superior usability despite potential performance trade-offs. The ease of integration and the reduced cognitive load for developers translate to faster development cycles and improved maintainability—critical factors for the long-term success of the sys-sage library's Python interface.

Furthermore, pybind11's robust support for C++ features, including STL containers and inheritance, aligns perfectly with the sys-sage library's complexity. Its ability to seamlessly handle these features simplifies the wrapping process and ensures that the Python interface accurately reflects the C++ library's functionality. This comprehensive feature support, combined with its ease of use, makes pybind11 the optimal choice for creating a maintainable and extensible Python interface for the sys-sage library.

4 Implementation

4.1 Project Structure

The Python bindings for the sys_sage library are encapsulated within a dedicated wrapper file. This file serves as the primary interface between the C++ library and the Python environment, ensuring a structured and maintainable codebase. We can see the general file structure in Figure 4.1.

This structure is designed to provide a clear and organized layout, facilitating easy navigation and modification of the bindings.[14, see The Basics/First Steps]

The file is segmented into distinct sections, each serving a specific purpose:

- **Include Directives:** Essential header files, including those from pybind11 and the sys_sage library, are included to provide the necessary functionalities.
- **Global Variables and Helper Functions:** This section accommodates any global variables or helper functions required for the bindings.
- **PYBIND11_MODULE Macro:** This macro defines the Python module and its contents. Within this macro, the following elements are defined:
  - **DATA:** C++ macros are exposed to Python in form of module attributes, allowing Python code to use these as constants.
  - **Class Bindings:** C++ classes are bound to Python classes using `py::class_<>`. This includes defining constructors, methods, and properties.
  - **Module Functions:** C++ functions that are not class methods are bound to the Python module itself.

This structured approach ensures that the bindings are well-organized and easy to maintain, providing a solid foundation for the Python interface of the sys_sage library.

4.2 Object-Oriented Programming with pybind11

Since `sys-sage` is a perfect example for object-oriented programming we will focus on every key aspect of it in this section, involving classes, constructors and inheritance, as well as methods and attributes.

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include <pybind11/attr.h>
#include "sys-sage.hpp"

// Global variables and helper functions

PYBIND11_MODULE(sys_sage, m) {
    // Expose C++ Macros to Python
    m.attr("COMPONENT_NONE") = SYS_SAGE_COMPONENT_NONE;

    // Bind the Component class
    py::class_<...>(m, "Component", ..., "Generic Component")
        // Bind the constructor(s)
        .def(py::init<...>())
        // Bind member functions
        .def("InsertChild", &Component::InsertChild, ...)
        // Bind properties (getters and setters)
        .def_property("parent", &Component::GetParent, &Component::
            SetParent, ...);

    // Bind module-level functions
    m.def("parseCccbenchOutput", &parseCccbenchOutput, ...);
}
```

Figure 4.1: The structure of the wrapper file.

4.2.1 Class Definitions

Python and C++ both offer object-oriented programming. C++ differentiates between structs and classes, while Python primarily uses classes. pybind11 allows C++ structs and classes to be bound to act like Python classes. [14, see The Basics/Object-Oriented Code]

For each functionality, a table like Table 4.1 is provided for comparison.

To use `sys-sage` in C++, we need to include `syssage.hpp` in the project. To use `sys-sage` in Python, we need to include the `sys_sage` module, where we can also rename the module.

We can see that pybind11 bindings are straightforward. We use `py::class_<T>` where `T` is the template input, in this case, `Component`. `py::class_<T>` takes the module, class

C++ implementation	`class Component {`
	`  // ...`
	`};`

Pybind11-Binding	`py::class_<Component, std::unique_ptr<Component, py::nodelete>>(`
	`  m, "Component", "Generic␣Component"`
	`);`

C++ usage	`#include "sys-sage.hpp"` `Component;`
Python usage	`import sys_sage as sys_sage` `sys_sage.Component`

Table 4.1: Class Definition Comparison

name, and class documentation as arguments. As mentioned in the previous chapter, for pybind11 to indicate that component objects should not be destroyed by Python, we use `std::unique_ptr<T>` as holder type for the Component class, and we add a delete function, that is `py::nodelete`. There are also other methods to define how Python should handle the underlying C++ objects. [14, see The Basics/Object-Oriented Code]

4.2.2 Constructor

If we would define a class in native Python, it might look like this:

```
class Example(self):
    def __init__(self, id, name):
        self.id = id
        self.name = name
```

With pybind11, we can take already defined Constructors of C++ implementation and can use the convenience function `init<>()` that binds automatically the C++ Constructors. [14, see The Basics/Object-Oriented Code]

C++ implementation provides a bunch of Constructors for Component-Class and Subclasses. In our use case, pybind11's convenience function is indeed convenient.

For all functions and properties we want to bind to, we use `.def(...)`. That also applies to initialization functions, where we use the function `py::init<T>()`. As one can see, this is a template function, and we replace `T` with all the types of the Arguments there are defined in the C++ Constructor. In our example case (4.2), we have types `int`, `string`, `int` in this order. [14, see The Basics/Object-Oriented Code]

C++ implementation	`Component(`
	`  int id = 0,`
	`  string name = "unknown",`
	`  int componentType = SYS_SAGE_COMPONENT_NONE`
	`);`

Pybind11-Binding	`.def(`
	`  py::init<int, string, int>(),`
	`  py::arg("id") = 0,`
	`  py::arg("name") = "unknown",`
	`  py::arg("type") = SYS_SAGE_COMPONENT_NONE`
	`)`

C++ usage	`new Component();`
Python usage	`sys_sage.Component()`

Table 4.2: Constructor Comparison

As in every definition of a function in the bindings, we can use `py::arg` to not only define default values but also we can give the argument names for the documentation. In this case, this might look like this: `py::arg("id") = 0`. Instead of this, there is also a shorthand notation that would look like this: `"id"_a=0`. However, the literal operator must be made visible before with `using namespace pybind11::literals`. [14, see Advanced Topics/Functions]

```
// Component-constructor without parent
.def(
    py::init<int, string, int>(),
    py::arg("id") = 0,
    py::arg("name") = "unknown",
    py::arg("type") = SYS_SAGE_COMPONENT_NONE
)

// Component-constructor with parent
.def(
    py::init<Component *, int, string, int>(),
    py::arg("parent"),
    py::arg("id") = 0,
    py::arg("name") = "unknown",
    py::arg("type") = SYS_SAGE_COMPONENT_NONE
```

```
)
```

4.2.3 Inheritance

If we had to define Inheritance in Python, we can do it like this:

```python
class Vehicle:
    def honk(self):
        print("Honk!")

class Car(Vehicle):
    def drive(self):
        print("Vroom!")

my_car = Car()
my_car.honk()
my_car.drive()
```

The sys_sage library has Component as a generic class and many Component-Types as Subclasses. Inheritance should work properly, and indeed pybind11 supports that.

C++ implementation	`class Topology : public Component {`
	`  // ...`
	`};`

Pybind11-Binding	`py::class_<`
	`  Topology,`
	`  std::unique_ptr<Topology, py::nodelete>,`
	`  Component`
	`>(m, "Topology")`

C++ usage e.g.	`t.GetId(); // t is Topology, but can access Component's`
	`  // attributes and functions`

Python usage	`t.id # Works as in C++`

Table 4.3: Inheritance Comparison

The super-class is given as input in the template of py::class_<>. pybind11 will automatically care that all the functions and attributes from the super class are accessible to the Subclass. Multiple inheritance is also allowed. This is not yet needed but maybe in the future. [14, see The Basics/Object-Oriented Code]

4.2.4 Functions

With pybind11, binding functions is quite simple. Consider this sys-sage function for example:

C++ implementation	`void Component::InsertChild(Component * child){`
	`  // ...`
	`}`

Pybind11-Binding	`.def(`
	`  "InsertChild",`
	`  &Component::InsertChild,`
	`  py::arg("child")`
	`)`

C++ usage e.g.	`c->InsertChild(child);`
Python usage	`c.InsertChild(child)`

Table 4.4: Function Comparison

In `.def()`, the first argument is the name of the method. Note, that normally in Python, the convention is to use snake-case for a method-name. In our case, we use the same name as in the C++ implementation. For some other specific methods, we use simple naming, but more on this in the Attribut-section. The next argument is the C++ function to bind to, and the last argument is the argument name for the function. This is not necessary, but it helps for documentation and allows usage of keyword arguments. [14, see Advanced Topics/Functions]

Some functions in `sys-sage` are overloaded. We now take a look at how to bind overloaded functions.

4.2.5 Overloaded methods

To use overloaded methods, we need to pick up the method signature to map the correct overloaded method.

C++ implementation	`void Component::PrintSubtree() {`
	`  PrintSubtree(0);`
	`}`
	
	`void Component::PrintSubtree(int level) {`
	`  PrintSubtree(0);`
	`}`

Pybind11-Binding	`.def(`
	`  "PrintSubtree",`
	`  (void (Component::)()) &Component::PrintSubtree,`
	`  "Print␣the␣subtree␣of␣the␣component"`
	`)`
	
	`.def(`
	`  "PrintSubtree",`
	`  (void (Component::)(int)) &Component::PrintSubtree,`
	`  "Print␣the␣subtree␣..."`
	`)`

C++ usage e.g.	`c.PrintSubtree();` `c.PrintSubtree(0);`
Python usage	`c.PrintSubtree()` `c.PrintSubtree(0)`

Table 4.5: Overloaded Methods Comparison

The only difference to "normal" binding is that with overloading, we add the method signature, which is in this case `void (Component::) ()` and `void (Component::) (int)` respectively, which we add before the target method. [14, see The Basics/Object-Oriented Code]

In some cases, we don't have any C++ function we can bind to. This is because either we want to add some additional methods or there is some logic needed before the C++ function can be called. For these cases we can use lambda functions.

4.2.6 Lambda functions

Lambda functions are usefull, if we want to add functionality, that should only be available for the Python implementation of `sys-sage` or we need to add some operations and logic before a specific `sys-sage` function can be called.

For the first case, we have this example:

```
.def(
    "__bool__",
    [](Component\& self){
        std::vector<Component*> children = self.GetChildren();
        return !children->empty();
    }
)
```

Here, we define the bool-function, which is called whenever we want to extract a bool value from an object of any type, in this case Component-type. Within the lambda function, we check if the Component object has any children. This is convenient because we can then do the following in Python:

```
comp = sys_sage.Component()
# ...
if comp:
    # comp has children
else:
    # comp has no children
```

So this is an example for an extra feature, that is not present in the base implementation of `sys_sage` in C++. But this also shows that there it is quite straightforward to add more functionalities with a few lines of code. Later, when we get to the XML I/O functionalities, we will see another example of lambda functions in method definitions.

We can see that we can bind class-methods, however we also have some functions in `sys-sage` that are not to be accessed over an object.

4.2.7 Module functions

`Sys-sage` provides parsers, that are not part of any class, as some kind of class methods. Rather they stand alone, making it necessary to bind them as module functions.

So this is quite similar to the class methods, but here we assign the function to the module interface object `m`.[14, see Reference/Conveneince Classes for specific Python types]

C++ implementation	`int parseCapsNumaBenchmark(`
	`  Component* rootComponent,`
	`  string benchmarkPath,`
	`  string delim`
	`){`
	`  // ...`
	`}`

Pybind11-Binding	`m.def(`
	`  "parseCapsNumaBenchmark",`
	`  \&parseCapsNumaBenchmark,`
	`  py::arg("root"),`
	`  py::arg("benchmarkPath"),`
	`  py::arg("delim") = ";"`
	`);`

C++ usage e.g.	`parseCapsNumaBenchmark( root, ‘‘benchmark.csv’’, ‘‘;’’);`
Python usage	`sys_sage.parseCapsNumaBenchmark(root, "benchmark.csv", ";")`

Table 4.6: Module Function Comparison

We showed that all kinds of functions defined in `sys-sage` can be bound to Python, leaving only attributes to find a way to bind them.

4.3 Attributes

This section delves into the intricacies of attributes within the `sys_sage` library, expanding on the object-oriented programming concepts discussed earlier. Attributes warrant a dedicated discussion due to the numerous special cases and module-specific considerations.

4.3.1 Module Attributes

`sys_sage` employs macros in its C++ implementation to define component and datapath types. To maintain consistency and offer similar functionality in Python, we utilize module attributes.

In C++, these macros are preprocessed and replaced with their corresponding integer values. In our solution, they are treated as exported variables, registered using the `attr` function, effectively making them module attributes. pybind11 handles the necessary type casting, as all macros resolve to integers.[14, see The Basics/First Steps]

C++ implementation	`#define SYS_SAGE_COMPONENT_NONE 1`
Pybind11-Binding	`m.attr("COMPONENT_NONE") = SYS_SAGE_COMPONENT_NONE;`
C++ usage e.g.	`c.CountAllSubcomponentsByType(SYS_SAGE_COMPONENT_NONE);`
Python usage	`c.CountAllSubcomponentsByType(sys_sage.COMPONENT_NONE)`

Table 4.7: Module Attribute Comparison

A naming convention is employed where the "SYS_SAGE" prefix is omitted from the Python attribute names to avoid redundancy. As demonstrated in the example, the attributes are accessed via `sys_sage.COMPONENT_NONE`. A significant advantage of the Python implementation is its enhanced documentation capabilities. A user's Integrated Development Environment (IDE) or the `help(sys_sage)` function in python can display all module attributes within the `DATA` section, aiding users in selecting the appropriate options.

4.3.2 Properties (Default Attributes)

In C++, class attributes are typically private and accessed through setter and getter methods. However, this pattern is not idiomatic in Python. pybind11 offers the `def_property` function to bind setter and getter methods to Python properties, providing a more natural syntax. [14, see The Basics/Object-Oriented Code]

C++ implementation	`class Component{`
	`  private:`
	`  string name;`
	`};`
	`...`
	`void SetName(string name){...}`
	`string GetName(){...}`

Pybind11-Binding	`.def_property("name", &Component::GetName, &Component::SetName)`
C++ usage e.g.	`string name = c.GetName();` `c.SetName("name");`
Python usage	`name = c.name` `c.name = "name"`

Table 4.8: Property Comparison

This syntax might be more intuitive for Python users. pybind11 also provides `def_property_readonly` for read-only properties, such as the `id` attribute.[14, see The Basics/Object-Oriented Code]

C++ implementation	`class Component{`
	`  private:`
	`  int id;`
	`};`
	`...`
	`string GetId(){...}`

Pybind11-Binding	`.def_property_readonly("id", &Component::GetId)`
C++ usage e.g.	`int id = c.GetId();`
Python usage	`id = c.id`

Table 4.9: Read-only Property Comparison

4.3.3 Dynamic Attributes

Python natively supports dynamic attributes, allowing users to add attributes to objects at runtime.

```
class Dog():
    # no attributes defined here
    ...

d = Dog()
d.hungry = True
d.hungry # True
```

This behavior is desirable for `sys_sage` components and datapaths, especially considering the C++ implementation's `std::map<std::string, void*> attrib` which facilitates dynamic attribute storage.

Attribute-Map

The `attrib` map offers high flexibility and performance by storing `void*` pointers with string keys. The idea behind our Python implementation includes storing both C++ types for the default attributes and Python types for custom attributes in the same

attrib map. Users should be able to modify these attributes, mirroring the C++ base implementation.

There are multiple possibilities to expose the attrib map to Python. For example, pybind11 can activate Python's native dynamic attribute functionality using py::dynamic_attr() as an extra flag in the class definition. [14, see The Basics/Object-Oriented Code]

```
py::class_<Component, std::unique_ptr<Component, py::nodelete>>(m,
    "Component", py::dynamic_attr(),"Generic Component");
```

This enables dynamic attribute usage in Python:

```
c = sys_sage.Component()
c.dynamic = "attribute"
c.dynamic # returns "attribute"
```

If we want to conntect the attrib map to Python attributes we need to override the __setattr__ and __getattr__ methods, but this breaks property functionality. Handling properties like name or id in custom setter and getter methods, that we define, would work as well, but results in bloated code. Therefore, an alternative solution is considered. This approach uses the [] operator to access attributes, overriding __getitem__ and __setitem__.[14, see Reference/Convenience classes for arbitrary Python Types][10]

This also helps to have a clear destinction between basic and custom attributes, similar to the C++ implementation.

```
c = sys_sage.Component()
# solution with dynamic attributes
c.example = "test" # calls __setattr__
c.example # calls __getattr__
# solution with items
c["example"] = "test" # calls __setitem__
c["example"] # calls __getitem__
```

Overloaded __getitem__ methods allow access by key and index. The following example demonstrates this:

```
c["example"] = "test"
c[0] # returns "test" if "example" is the only custom attribute
```

In the Python implementation, accessing and modifying attributes is designed to be user-friendly. The getter and setter methods for attributes are implemented to handle

Python objects of any type or default attributes. This abstraction allows users to interact with custom attributes without explicit type casting. For default attributes, users need to be aware of the corresponding Python types.

In contrast, the C++ implementation requires more explicit memory management and type handling for attribute manipulation. The following code snippet illustrates the process of updating and retrieving an attribute in C++:

```
// Update Attribute
std::string key = "example";
if (c->attrib.find(key) != c->attrib.end()) {
    std::string* oldValue = static_cast<std::string*>(c->attrib[
        key]);
    delete oldValue; // Free the old value
}
std::string* newValue = new std::string("test"); // Allocate new
    value
c->attrib[key] = static_cast<void*>(newValue); // Update the map

// Retrieve Attribute
if (c->attrib.find(key) != c->attrib.end()) {
    std::string* retrievedValue = static_cast<std::string*>(c->
        attrib[key]);
    std::cout << "Value: " << *retrievedValue << std::endl;
} else {
    std::cout << "Key not found." << std::endl;
}
```

As demonstrated in the code, updating an attribute in C++ involves checking for the existence of the key, explicitly freeing the memory associated with the old value, allocating memory for the new value, and then updating the attribute map with a void pointer. Similarly, retrieving an attribute requires casting the void pointer back to the expected type after verifying the key's existence.[22]

This manual memory management and type casting in C++ contrasts with our Python implementation, where we always expect Python objects from and to the user side. Therefore the user does not need to worry about any of these memory management related issues. In the following we briefly explain the helper functions that are used:

The `set_attribute` function examines default values, in which case it converts data to C++ types, or it encapsulates Python data types in shared pointers, if it is not a default value. It then stores them in the `attrib` map as void pointers. The `get_attribute` function retrieves attributes using a key or an index and casts the `void*` pointers to

their appropriate types. The `del_attribute` function safely removes attributes from the map.

For the Component class the second approach, using items, was chosen for its simplicity and compatibility with existing properties. For datapaths however dynamic attributes were enabled. Since the `attrib` map is not being used in the xml export and import functionality, we use the native Python dynamic attributes for datapaths' attributes. Speaking of XML I/O we will now see how the export works in `sys-sage` in Python. Since the import functionality is introduced with this work we will provide more details on this new feature.

4.4 XML Export

4.4.1 Original Implemented Functionality

The base C++ implementation of `sys-sage` includes XML export functionality, which traverses the topology and generates an XML representation of it. XML's hierarchical structure is well-suited for representing topologies, where the topology itself is the root node, cores and threads are leaf nodes, and various components reside in between.

XML properties are utilized to represent default attributes such as `id`, `name`, `type`, and type-bound attributes like `size` for component's subclass `Memory`. Datapaths are also included, with attributes like `src`, `target`, `orientation`, `latency`, and `bandwidth`.

Dynamic attributes, which users can add in C++, are represented as separate XML nodes:

```
<Attribute name="GPU_Clock_Rate">
    <GPU_Clock_Rate frequency="2.000000" unit="MHz"/>
</Attribute>
```

Since the `attrib` map in C++ stores attributes as `void*`, the XML export function requires user-defined functions to parse custom attribute types. Users can define up to two functions:

```
std::function<int(string,void*,string*)>
    search_custom_attrib_key_fcn;
std::function<int(string,void*,xmlNodePtr)>
    search_custom_complex_attrib_key_fcn;
```

The first function must return a string representation of the attribute, while the second allows for creating more complex XML nodes. The export function can be called as follows:

```
exportToXml(
    root,
    path,
    search_custom_attrib_key_fcn,
    search_custom_complex_attrib_key_fcn
);
```

4.4.2 Process Description

The Python implementation of `exportToXml` mirrors the C++ functionality, allowing users to define custom parsing functions as in this example:

```
def search_custom(key: str, value) -> str:
    if "example" in key:
        return str(value)+"test"
    return None

def search_custom_complex(key: str, value) -> str:
    return "<root><Attribute␣key=\"" + str(key) + "\"␣value=\"" +
        str(value) + "\"/><root>"
```

The `key` is always a string, while `value` can be any Python type. The return value must be a string, representing either a simple value or an XML node respectively. When creating the XML node string for complex attributes the target XML information is incapsulated in a tag that is ommitted in the parsing operation. XML parsing modules like `xml.etree.ElementTree`[25] can be used to construct complex XML nodes in Python.

The Python export function can be called as follows:

```
sys_sage.exportToXml(root, path, search_custom,
    search_custom_complex)
```

Global variables are used to store the user-defined functions. Two helper functions are used to translate between Python and C++ types. The helper functions implement the required functions with the correct signature for the XML-export function and within the helper functions the user-defined Python functions - previously stored in the global variables - are called to achieve the desired functionality. Default attributes are skipped, but this can be changed in the future.

4.4.3 Helper Functions

In the following we explain which steps are performed by the helper functions:

Simple Attributes

1. Check if the attribute is a default attribute. If yes, skip it.
2. Cast the `void*` to a Python object.
3. Call the user-defined Python function, passing the key and value (Python object).
4. Cast the returned Python string to a C++ string.
5. Store the string in the return string pointer.

Complex Attributes

1. Follow steps 1-3 for simple attributes.
2. Expect user-defined function to return XML node as a Python string.
3. Convert Python string to C++ string.
4. Parse the XML node using `xmlParseDoc` [24] and add it to the XML node.

Now that we've seen how the `exportToXml` function works we continue with the `importFromXml` function.

4.5 XML Import

The XML import function mirrors the XML export function, operating in reverse. This section provides a detailed explanation of the import function's inner workings, as it represents a new feature in the C++ base implementation.

4.5.1 Process Description

The XML import process involves the following steps:

1. **Read XML File:** The function begins by reading the input XML file.
2. **Find Component and Datapath Nodes:** It then locates component and datapath nodes within the XML structure.

3. **Traverse Components:** The function recursively traverses all components, adding them to a map of addresses (obtained from the XML file) and newly created component instances.

4. **Attribute Collection:** Upon encountering an attribute node, the function calls an attribute collection routine. This routine attempts to parse attributes using the following order:

   a) Custom function for simple attributes.

   b) Default function for simple attributes.

   c) Custom function for complex attributes.

   d) Default function for complex attributes.

   If no parsing method succeeds, the attribute is omitted.

5. **Add Datapaths:** Datapaths are created by inspecting the component map, which stores components and their corresponding addresses from the XML file. This ensures correct connection of datapaths to the created components.

6. **Return Root Component:** The function returns the root component, which is of type Topology.

4.5.2 User-Defined Functions

Users can provide custom functions for attribute parsing:

```
std::function<void*(xmlNodePtr)> search_custom_attrib_key_fcn =
    NULL;
```

The function for simple attributes receives an `xmlNodePtr` and returns a `void*` to store the attribute value in the `attrib` map.

```
std::function<int(xmlNodePtr, Component *)>
    search_custom_complex_attrib_key_fcn = NULL;
```

The function for complex attributes receives an `xmlNodePtr` and a pointer to the current component. It allows users to configure component properties directly via the `attrib` map. The function must return 1 for success and 0 for failure.

Similar to the export function the import function can be called like this:

```
Component* importFromXml(
    path,
    search_custom_attrib_key_fcn,
```

```
    search_custom_complex_attrib_key_fcn
);
```

4.5.3 Python Interface

Users can provide up to two functions in Python. Here is an example of how this can be done:

Simple Attributes:

```
def search_custom(x : str):
    if "example" in x:
        y = x.split("value=\"")[-1]
        return y.rstrip("\"/>")
    else:
        return None
```

Complex Attributes:

```
def search_custom_complex(x : str,c : Component) -> int:
    x = str(x)
    m = re.search(r'<my_core_info\s+temperature="(\d*\.\d*)"␣
        temp_unit="(\w*)"␣frequency="(\d*)"␣freq_unit="(\w*)"/>',
        x)
    if m:
        my_core_info = {}
        my_core_info["temperature"] = float(m.group(1))
        my_core_info["temp_unit"] = m.group(2)
        my_core_info["frequency"] = int(m.group(3))
        my_core_info["freq_unit"] = m.group(4)
        c["my_core_info"] = my_core_info
        return 1
    else:
        return 0
```

The `importFromXml()` function can be called as follows:

```
sys_sage.importFromXml(path, search_custom, search_custom_complex)
```

Figure 4.2 illustrates the steps, that are involved with the import functionality in the Python implementation. In the C++ implementation the user can directly define own

functions for parsing the attributes. These fuctions are called from the `collect_attrib` function. In contrast to the Python implementation some extra steps are necessary. We've already seen this approach in the export function, where some global variables store the user-defined Python functions. Helper functions facilitate type conversion between Python and C++. In Figure 4.3 we can examine the inner workings of the `collect_attrib` function, where the helper functions are called. In the next section the exact steps followed in these helper functions are explained.

4.5.4 Helper Functions

Simple attributes:

1. Dump the XML node content to a string.
2. Cast the string to a Python string.
3. Call the user-defined Python function with the Python string.
4. Store the returned Python object in the `attrib` map.

Complex attributes:

1. Perform the same 2 steps as for simple attributes.
2. Pass the string and the current component to the Python function.
3. The Python function returns a success status (1 or 0).

Same as in export functionality we need the helper functions to implement the required functions with the proper signature. It is safe to say, that the import functionality gives the user more freedom, as he can directly access the component.

After implementing the XML I/O functions for the Python version of `sys-sage`, we can now introduce one of the challenges, that we're facing.

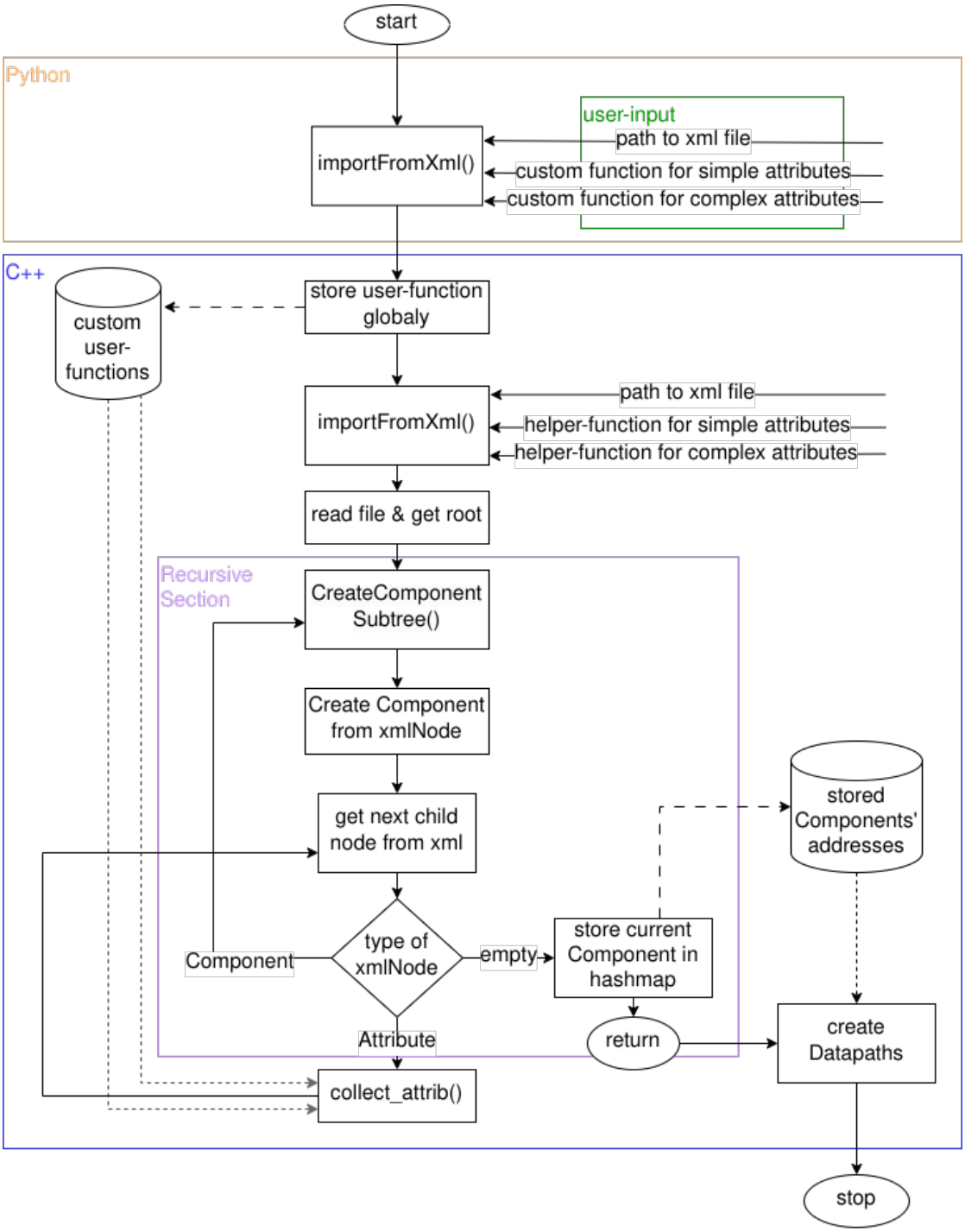


Figure 4.2: XML Import Flowchart

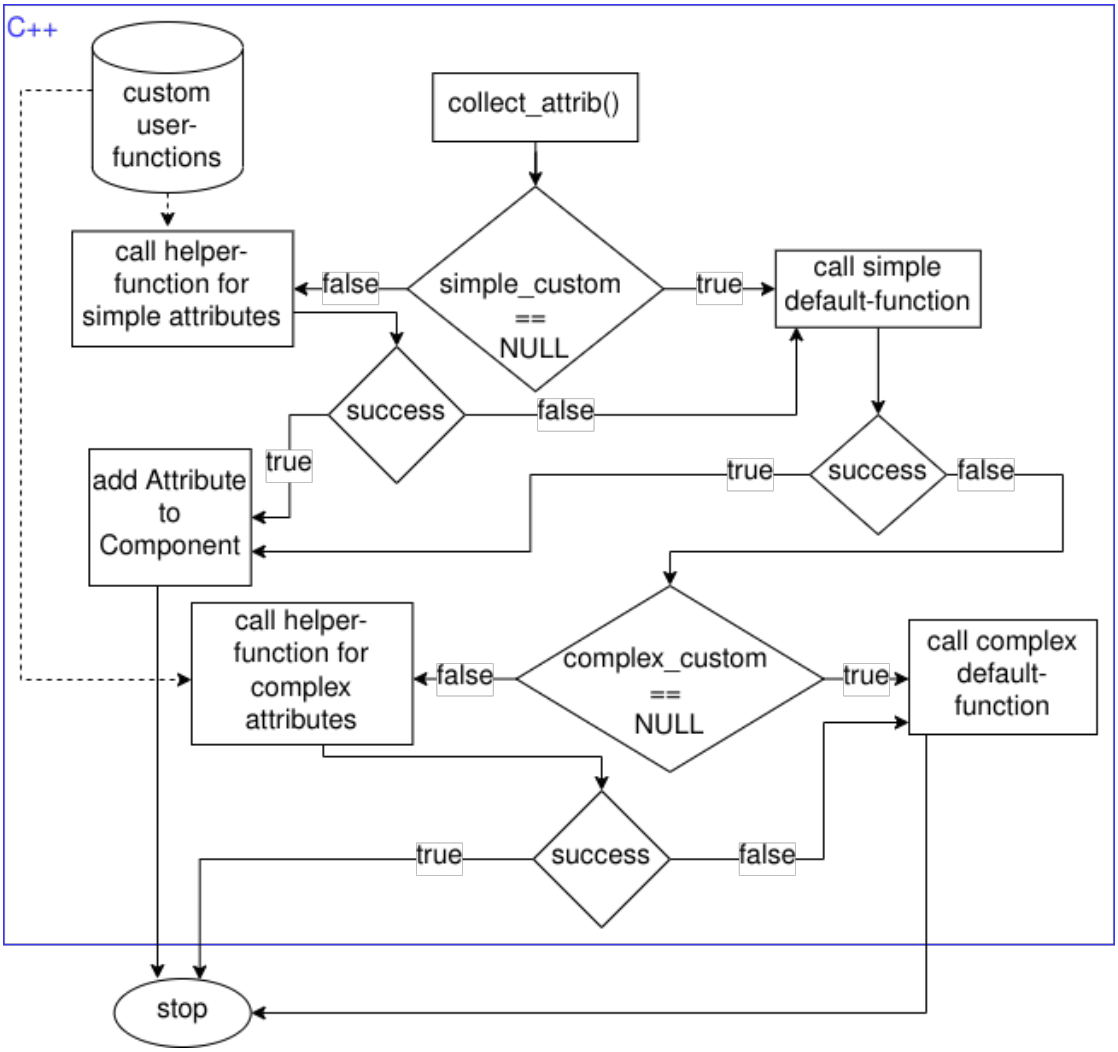


Figure 4.3: Attribute parsing

4.6 Ownership Management

A critical challenge in interfacing C++ with Python involves the management of object ownership, particularly concerning the `Component` objects within the `sys-sage` library. By default, pybind11 transfers ownership of the underlying C++ object to the Python side. This implies that when a Python object, such as `c = sys_sage.Component()`, is created, Python assumes responsibility for its lifecycle, including reference counting and deallocation. While this approach is generally adequate for objects without external dependencies, it becomes problematic when dealing with interconnected components, such as parent-child relationships or datapath associations.

Consider the following scenario:

```
c = sys_sage.Component()
child = sys_sage.Component(c) # Parent set in constructor
del c
d = child.parent
print(d.id) # Undefined behaviour
```

In this example, deleting `c` in Python leads to the deallocation of the corresponding C++ object. However, the `child` object retains a reference to the now-deleted parent. Consequently, accessing `child.parent` results in undefined behavior, as the referenced object no longer exists. This issue arises because the `Component` destructor does not update references in related objects, such as parent nodes, child nodes, or datapaths.

An initial attempt to address this problem involved implementing a custom delete function, `delete(bool withSubtree)`, to replace the default destructor. However, this approach was deemed unsuitable due to potential conflicts with Python's garbage collection mechanism. As noted in pybind11's documentation, overriding the `__del__(self)` method in Python is strongly discouraged, as it can interfere with the garbage collection process. [9][13]

To mitigate these issues, pybind11's capability to disable ownership transfer was utilized. By modifying the template parameters of the `py::class_<>` declaration, ownership of the C++ objects can be retained by the C++ side. Specifically, the `std::unique_ptr<Component, py::nodelete>` template parameter was employed. The `py::nodelete` keyword instructs pybind11 to prevent Python from deallocating the underlying C++ object. This ensures that the C++ side remains responsible for managing object lifetimes. [14, see Advanced Topics/Clasees]

However, this solution introduces a new challenge. The base C++ program provides methods for manually deleting objects, which, when exposed to Python, can lead to inconsistencies. For instance:

```
import sys_sage

c = sys_sage.Component()
d = c
c.Delete()
d.Delete() #Segmentation Fault
```

In this scenario, deleting `c` using the `Delete()` method deallocates the C++ object, but the Python wrapper `d` remains unaware of this change. Ideally, all references to the deleted object should be updated. Unfortunately, pybind11 does not provide a mechanism for automatically updating these references. While this scenario is relatively rare, it underscores the need for careful consideration of object deletion strategies.

As mentioned before, pybind11, by default, invokes the destructor of the underlying C++ object when the reference count reaches zero. To enhance consistency, modifying the destructor to update relevant references is considered.

4.6.1 Destructor modifications

In the base implementation, the destructors remain untouched. Consequently, to free resources, the user must explicitly call the `Delete` function. This function provides the option to also delete the subtree, encompassing all descendants of the component. Naturally, all Datapaths connected to the component are deleted in this process. However, the attributes stored in the `attrib` map are not automatically deallocated. Typically, in most containers from the STL, memory is freed upon destruction for all contained items except pointers.[7]

As an alternative to the approach requiring explicit `Delete` calls, thereby passing memory management to the C++ side, we now examine a solution utilizing custom destructors for the Component and Datapath classes. The following illustration, Figure 4.4, may help to grasp the details:

To remove the "Parent" in this Topology, we no longer call the `Delete` method. Instead, `delete Parent` is used. In this solution, the Destructor removes all Datapaths connected to the Component. Additionally, Child 1 and 2, as well as the Grandparent, are updated: the children's `parent` values are set to `NULL`, and Parent is removed from the Grandparent's children list. The children are no longer directly connected to their grandparent and can only be reached if a Datapath to another component remains. In our case, Child 1 becomes unreachable, as its only two connections — once in the Parent's children list and once via the Datapath to Parent — are destroyed with the deletion of Parent. This might also cause memory leaks if components fall out of scope without any remaining connection to the rest of the topology. Therefore, the

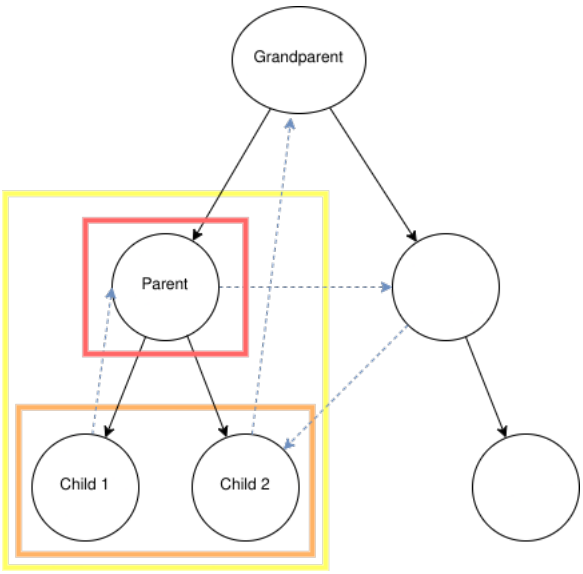


Figure 4.4: Topology Example

user should always consider calling the `DeleteSubtree` method beforehand, which has slight modifications compared to the base implementation but effectively achieves the same outcome.

The Datapath's destructor is also modified. Essentially, the logic behind the `DeleteDatapath` method is migrated to the destructor.

The solution presented in this section is cleaner and allows for seamless usage of the pybind11 binding, enabling the Python interpreter to manage memory deallocation.

In the next chapter, we evaluate our findings and the solutions, comparing the performance of the Python and C++ implementations.

5 Evaluation

The following evaluation chapter assesses the usability and performance of the developed Python interface for `sys-sage` against its C++ counterpart. It also examines the integration of the new XML import feature. Through benchmarks and qualitative analysis, we quantify the performance impact of the Python bindings and discuss the trade-offs between user-friendliness and efficiency. The evaluation mainly focuses on the Python implementation without the modifications to the destructors of the base implementation, therefore we discuss the strengths and weaknesses of the solution with the mentioned modifications.

5.1 Requirement Analysis and Performance Evaluation

First we want to revisit the requirements of the Python implementation of sys-sage, which are described in Chapter 3, followed by a performance evaluation of the Python and C++ implementations.

5.1.1 Library Complexity and Dependencies

The Python solution mirrors the C++ implementation, encompassing nearly all functionalities. The only current limitation is the handling of overloaded functions that accept a list of components as arguments. Attributes can be stored as Python attributes, and both import and export functionalities are available, with optional support for custom Python functions to parse the attributes. The basic setup, excluding optional dependencies like `intel_pqos` and `nvidia_mig`, requires only two dependencies: Python and pybind11.

5.1.2 Error Handling and Documentation

Simple error messages have been integrated, but there is room for improvement in error handling and user feedback. Basic documentation is provided, outlining the functionality of each function.

5.1.3 Future Support and Build/Installation

Pybind11 is a well-established tool for creating bindings, supported by an active community and ongoing maintenance. CMAKE support allows users to build either the Python or C++ version of sys-sage, providing flexibility in deployment.

5.1.4 User Experience

The Python interface offers a more user-friendly experience, enabling rapid prototyping and testing. For example, scheduling processes on HPC systems can be quickly prototyped and analyzed using Python scripts. If performance becomes a bottleneck, the solution can be transitioned to C++ for optimal execution. Memory management, however, requires careful attention, as users must manually manage subtree deletion to avoid undefined behavior. This work presents improvements to destructors, though memory management remains a complex aspect.

5.1.5 Performance Benchmarking

Setup and Methodology

To evaluate performance, C++ benchmarks utilize the high-resolution clock, while Python benchmarks employ the `timeit` module. The testing environment consisted of an Arch Linux system with an Intel i7-9750H CPU and 15GB of RAM.

It is important to acknowledge that precise benchmarking requires rigorous methodology, including accurate resource measurement, reliable process termination, deliberate core assignment, consideration of non-uniform memory access, avoidance of swapping, and isolation of individual runs. [6]

However, given the substantial performance disparities between Python and C++, the benchmarks presented herein provide a general overview. For use cases demanding high precision, adhering to established benchmarking standards is recommended.

The goal of this section is to compare the Python and C++ implementations to provide insights into the performance trade-offs associated with each language. As presented in the Chapter 4 we have two different solutions for the Python implementation of sys-sage. For the evaluation we will only consider the solution without destructor modifications.

Basic Functionality Performance

Function	Python (ns)	C++ (ns)	Factor
Parsing HWloc	2880354	1240570	2.32
Parsing capsnuma	4881208	365801	13.34
Parsing mt4g	71169350	52992073	1.34
Getting HWloc subtree nodelist	453509	14055	32.27
Getting mt4g subtree nodelist	26648203	589624	45.20
Getting all components	39845198	470893	84.62
Getting numa max bw	30936	70	441.94
Creating new component	17054	364	46.85

The results indicate that parsing functions exhibit comparable performance. However, get operations, particularly retrieving all components, are significantly slower in Python due to the overhead of converting `std::vector` to Python lists. Creating new components is also considerably more expensive in Python, attributed to the inherent overhead of wrapping C++ objects for Python. The benchmarks to get the datapath with the max bandwidth from a component, which involves additional logic beyond a simple API call, demonstrates a substantial performance difference, highlighting the cost of loops in Python.

XML Import and Export Performance

Function	Python (ns)	C++ (ns)	Factor
XML Import			
Both functions	1859323	387664	4.80
Only simple	175492	39347	4.46
Only complex	251298	35731	7.03
None	94179	37559	2.51
XML Export			
Both functions	209590	53481	3.92
Only simple	175346	50376	3.48
Only complex	225953	50914	4.44
None	163335	51095	3.20

The XML import and export benchmarks, conducted with two components, where one of them has two attributes (one simple, one complex), reveal that the Python implementation with only complex attribute user functions is more costly. The C++ implementation, which directly writes to `xmlNodePtr`, is faster than the Python counterpart, which involves string conversions and parsing.

Attribute Access Performance

Operation	Python (ns)	C++ (ns)	Factor
Retrieve Attribute	4054	182	22.28
Update Attribute	4246	664	6.39

Attribute retrieval is faster in C++, while attribute updates involve additional steps (key existence check, memory freeing), resulting in higher overhead.

5.1.6 Python implementation comparison with and without destructor modifications

In Chapter 4, certain limitations of the initial implementation were discussed, particularly the lack of Python's ownership over created objects. The primary concerns involve potential memory leaks and an increased burden on the user, which deviates from typical Python application paradigms.

Consequently, an alternative approach employing modified destructors was explored to enable the Python interpreter to effectively manage memory deallocation. This obviates the need for manual object deletion by the user. However, this strategy introduces a potential issue: objects might be inadvertently deleted if they fall out of scope prematurely, even if they should persist within the topology. This scenario is most likely to occur when parsers are used to construct the topology, but the created objects are not subsequently registered on the Python side. In most practical use cases, objects are registered as soon as a function is called to retrieve a component from the topology. While this specific problem warrants future attention, the overall approach of utilizing modified destructors appears more promising for seamless integration with Python's memory management.

6 Related Work

6.1 Using pybind11 with OpenFOAM

OpenFOAM is a popular software package used for Computational Fluid Dynamics (CFD). A growing need exists to integrate Python-based machine learning with C++-based OpenFOAM, but current methods require translating Python code or models, which complicates development. [16]

The paper by Simon Rodriguez and Philip Cardiff presents a general approach to execute Python code directly within OpenFOAM using the pybind11 library, avoiding the need for translation. pybind11 is a lightweight library that enables embedding a Python interpreter within a C++ application. The workflow involves: Beginning the OpenFOAM execution, creating a Python interpreter within OpenFOAM, transferring data between OpenFOAM and the interpreter, executing Python code from within OpenFOAM and returning results to OpenFOAM. Regarding data transfer, pybind11 allows copying data between OpenFOAM and Python, and for efficient transfer of large fields, data can be passed as NumPy arrays. Data can also be exchanged by reference for improved performance. Python scripts and functions can be loaded and executed from within OpenFOAM. [19]

Test cases demonstrate the approach in OpenFOAM through examples like Python velocity profile boundary conditions, heat transfer solver prototyping, and field calculations using Python with machine learning. The proposed approach is feasible and efficient, and passing data as entire fields by reference is the most efficient data transfer method. In conclusion, the presented approach enables the use of Python within OpenFOAM, particularly for machine learning applications, and this method can expand the use of Python-based solutions in OpenFOAM. [19]

6.2 Measuring Performance of Python-C++ Interfacing technologies

A reasonable part of this work is the comparison of the different technologies and their performance. In more detail these comparisons were made by Jevgeni Antonenko. He presents a benchmark comparing different tools for creating C++-Python bindings.

The benchmark evaluates the performance overhead of these tools, focusing on function calls and data wrapping/unwrapping, using Mandelbrot set generation and Conway's Game of Life simulations as test cases. The study compares both manual and automatic binding generators, as well as static and runtime binding approaches (including Cython, pybind11, nanobind, SWIG, CFFI, and cppyy).

The results highlight the performance differences between the tools, with manually created or static bindings generally exhibiting better performance, while runtime bindings and pybind11 show more overhead. Notably, the study also reveals that minimizing C++/Python interoperability, such as through single, comprehensive function calls, yields the best performance, regardless of the binding tool. [3]

6.3 Using pybind11 to create a python wrapper for homomorphic encryption

Alexander J. Titus and Shashwat Kishore introduced PySEAL, a Python wrapper for the SEAL homomorphic encryption library, which is implemented in C++. Homomorphic encryption allows operations on encrypted data, a technique with growing applications in bioinformatics for data privacy.

PySEAL aims to make homomorphic encryption more accessible to bioinformatics researchers by providing a Python interface to the SEAL library, facilitating rapid prototyping of bioinformatics pipelines that utilize encryption. The paper details the implementation of PySEAL using pybind11 and discusses its potential applications in areas such as encrypted mutation searching and secure E-commerce recommendation systems. [21]

7 Future Work and Conclusion

7.1 Future Work

7.1.1 Optimization Flags and Performance Improvements

The base implementation, while functional, lacks optimization flags, which significantly impacts performance. To enhance the efficiency of the C++ components, future iterations should incorporate appropriate compiler optimization flags as this would also improve the performance in the Python layer.

7.1.2 Smart Pointers vs. Raw Pointers

Transitioning from raw pointers to smart pointers offers a robust approach to memory management. This transition will mitigate the risk of resource leaks and streamline the allocation and deallocation of memory. Specifically, storing attribute values as smart pointers eliminates the need for manual deletion, enhancing code safety and maintainability.

7.1.3 Enhanced Destructor Usage

Elaborating on the use of destructors will further improve compatibility with smart pointers in future developments. This proactive approach ensures that resource cleanup is handled consistently and efficiently, regardless of the memory management strategy employed.

7.1.4 Permit default Attributes for custom parsing in XML I/O

In this work we restricted the user-defined functions to only parse attributes, that are custom attributes, defined by the user. As we already have functions for the default attribute parsing, this could be extended in the future if needed.

7.2 Conclusion

This thesis explored various technologies for integrating C++ with Python, with a particular focus on SWIG and Pybind11. While both technologies offer viable solutions, Pybind11 emerged as the preferred choice due to its superior documentation, steeper learning curve, and comprehensive feature set.

A significant enhancement implemented in this project is the addition of XML import functionality. This feature allows users to import XML data in both the C++ and Python versions of the application. Furthermore, users can define custom parsing functions for attributes, providing flexibility and control over the data ingestion process. This functionality represents a substantial improvement in the application's data handling capabilities.

Despite exhibiting a slight performance overhead, Pybind11 provides a user experience that closely resembles native C++ development. The performance considerations, particularly regarding data access, highlight the importance of careful design and implementation. For larger projects, the integration of system-level functionality may present challenges. However, for rapid prototyping and testing, Pybind11 proves to be an invaluable tool.

In summary, the chosen integration approach facilitates a seamless and efficient interaction between C++ and Python, making it a compelling choice for projects that require a balance of performance and development speed.

Abbreviations

IDE Integrated Development Environment

HPC High Performance Computing

GPU Graphics Processing Unit

CPU Central Processing Unit

NUMA Non-Uniform Memory Access

API Application Programming Interface

STL Standard Template Library

List of Figures

List of Tables

Software Used

Table 1: AI-Based Tools Used

AI-Based Tool	**Use Case**	**Scope**	**Remarks**	**Last date of access**
ChatGPT	Spell checking, grammar checking and wording	Entire work		9.4.2025
Gemini 2.0 Flash	Spell checking, grammar checking and wording	Entire work		9.4.2025
Windsurf 1.42.7	Code Generation	Chapter 3-5	Auto completion and documentation in Python and C++	9.4.2025

Bibliography

[1] D. Abrahams and S. Seefeld. *boost.python*. 2015.

[2] J. Anderson. *Python vs C++: Selecting the Right Tool for the Job*. https://realpython.com/python-vs-cpp/#threading-multiprocessing-and-async-io.

[3] J. Antonenko. *Benchmark of Python-C++ bindings*. https://yanto.fi/2022/09/benchmark-of-python-c-bindings/. Sept. 3, 2022.

[4] S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein, and G. Gellner. *Cython Documentation*.

[5] F. Beuke. *GitHut 2.0, A small place to discover languages in GitHub*. https://madnight.github.io/githut/#/pull_requests/2024/1. 2024.

[6] D. Beyer, S. Löwe, and P. Wendler. "Reliable benchmarking: requirements and solutions." In: *International Journal on Software Tools for Technology Transfer* 21.1 (2019), pp. 1–29.

[7] cppreference.com. *std::map:: map*. https://en.cppreference.com/w/cpp/container/map/~map. 2021.

[8] P. S. Foundation. *ctypes — A foreign function library for Python*. Apr. 2, 2025.

[9] P. S. Foundation. *gc — Garbage Collector interface*. https://docs.python.org/3/library/gc.html. Oct. 2019.

[10] P. S. Foundation. *Python 3.x documentation: Data model*. https://docs.python.org/3/reference/datamodel.html. Apr. 9, 2025.

[11] P. S. Foundation. *Python/C API Reference Manual*. https://docs.python.org/3/c-api/index.html. Apr. 2, 2025.

[12] P. Fua and K. Lis. "Comparing python, go, and C++ on the n-queens problem." In: *arXiv preprint arXiv:2001.02491* (2020).

[13] W. Jakob. *Overloading __del__ method*. https://github.com/pybind/pybind11/issues/327. 2016.

[14] W. Jakob. *pybind11 — Seamless operability between C++11 and Python*. 2017.

[15] D. Lion, A. Chiu, M. Stumm, and D. Yuan. "Investigating managed language runtime performance: Why {JavaScript} and python are 8x and 29x slower than c++, yet java and go can be faster?" In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 835–852.

[16] O. Ltd. *About OpenFOAM*. `https://openfoam.com`. 2025.

[17] *Non-uniform memory access*. `https://foldoc.org/Non-uniform+memory+access`. 2024.

[18] A. Rigo and M. Fijalkowski. *CFFI documentation*. 2018.

[19] S. Rodriguez and P. Cardiff. "A general approach for running Python codes in OpenFOAM using an embedded Pybind11 Python interpreter." In: *arXiv preprint arXiv:2203.16394* (2022).

[20] *SWIG and Python*. Oct. 7, 2024.

[21] A. J. Titus, S. Kishore, T. Stavish, S. M. Rogers, and K. Ni. "PySEAL: A Python wrapper implementation of the SEAL homomorphic encryption library." In: *arXiv preprint arXiv:1803.01891* (2018).

[22] S. Vanecek. *sys-sage Documentation*. `https://stepanvanecek.github.io/sys-sage/html/index.html`. 2024.

[23] S. Vanecek and M. Schulz. *sys-sage: A Unified Representation of Dynamic Topologies & Attributes on HPC Systems*. Garching, Germany, 2024.

[24] D. Veillard. *The XML C parser and toolkit of Gnome*.

[25] *xml.etree.ElementTree - The ElementTree XML API*. `https://docs.python.org/3/library/xml.etree.elementtree.html`. Apr. 8, 2025.