Modern FFI alternatives for high-performance scientific computing in hybrid language workflows*

Isabella Basso do Amaral
isabellabdoamaral@usp.br

Renato Cordeiro Ferreira
Mathematics and Statistics Institute (IME)
University of São Paulo
renatocf@ime.usp

Advisor: Alfredo Goldman Vel Lejbman
Mathematics and Statistics Institute (IME)
University of São Paulo
gold@ime.usp

Date: 2024-11-28

Abstract

Computer software has become an invaluable tool in our times, advancing the frontiers of what is deemed possible on every field, and that includes research. One of the most common ways to enable contemporary workloads relies on a hybrid language approach, where the end-user usually operates on a high-level language, such as Python, while lowering the implementation of critical paths to a systems programming language, which can be compiled and optimized for the target hardware. This can be a very daunting task for inexperienced developers, and is quite often prone to severe errors, possibly compromising the validity of the results, or even the safety of the system. In this research project, we aim to explore alternatives that can make the process less error-prone, and guidelines to ensure robustness and reproducibility – a pillar of the modern scientific method. We begin by reviewing recent literature that has tackled similar issues. Then, we present a case study of a modern numerical algorithm comparing Python-hybrid implementations in terms of complexity, performance, and maintainability.

Keywords: Hybrid language workflows, FFI, Zig, Rust, Python, NumPy, benchmarking, scientific computing, numerical methods

*This paper is a work in progress. Please do not cite without permission.

1. Introduction

The role computers have acquired in our times is undeniable: from taking us to the moon, to providing optimized routes for delivery; it has become a staple to modern human endeavors. In order to make real progress in any field, it is necessary to have the right tools at hand, and that includes software. Software is built on top of layers of abstraction, each one providing a higher-level interface to the developer. Generally speaking, this does not come without costs. It can be empirically shown that some of the so-called *zero-cost abstractions* are not free, as they can add overhead due to indirections. Thus we start our discussion from the ground up, with the hardware.

1.1. Hardware

Turing machines are the theoretical model of computation, and have been used to prove the limits of what can be computed in a finite, or reasonable, amount of time. The main incarnation of the Turing machine is the von Neumann architecture, which is the basis of most modern computers [1].

1.1.1. Von Neumann architecture

The von Neumann architecture is composed of [2]:

- processing unit, which executes arithmetic instructions using registers (memory slots)
- control unit, which maintains the state of the processing unit
- memory, which stores data and instructions
- input/output, which connects the computer to the outside world
- external mass storage[2]

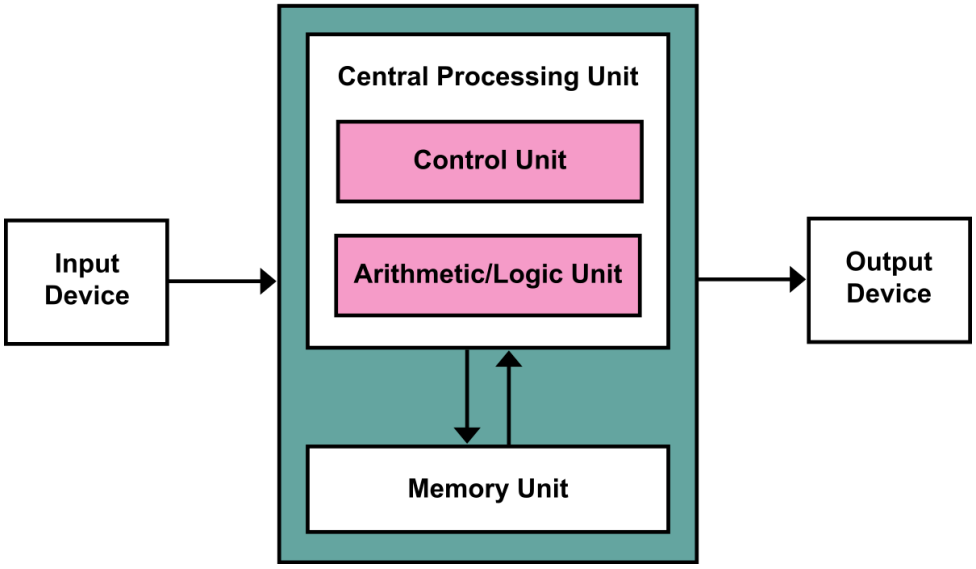

Figure 1: A von Neumann machine diagram. Source: Wikipedia.

1.1.2. Modern systems

In modern systems, the processing unit is coupled with the control unit, in what is commonly known as the CPU, or *Central Processing Unit*. The CPU has become hugely complex to enable speed-ups, being usually composed of several compute units, known as *cores*, which can act independently. As there is no single way to encode instructions, manufacturers have settled on common sets of instructions, known as

[2]Persistent (external mass) storage, such as hard drives or solid-state disks, is orders of magnitude slower than main memory, and is capable of storing data even when the computer is turned off. For our purposes, we only consider the main memory, which is volatile, and is used to store data that is being used by the system.

Instruction Set Architecture (ISA). The way those instructions are then executed is up to the manufacturer, and can vary greatly between different implementations of the same ISA. In general, the "control" unit is responsible for fetching the instructions from memory, decoding them and scheduling execution on a processing unit. This control unit is usually implemented with a sequencer circuit tied to a read-only memory (ROM) in the case of modern *AMD64* ISA implementations. In those implementations, after the instructions are fetched they are then sent for a queue to be matched to micro-ops (or hardware primitives) using *microcode* stored in the decoder ROM [3]. Each micro-op corresponds to processor primitives that can be scheduled for execution on a processing unit.[3] There are also optimizations such as pipelining and *out-of-order execution*, where the processor groups micro-ops that can be executed in parallel, or reorders them to maximize throughput [4].

The processor, however, is not the main bottleneck in most systems. The computer's main memory, called RAM or *Random Access Memory*, can be three orders of magnitude slower than the CPU, not only due to its slower clock speeds, but simply being at a couple of centimeters away of the processor already increases latency significantly. While some manufacturers have tried addressing the physical distance, the issue is still so pertinent that it is common to make use of *cache* memory, which is not only physically closer to the CPU, but also optimized for speed. The main issue with cache is that it can be very expensive, so it is usually small. Cache has become so important that multi-core architectures will usually have a hierarchy of caches, with the L1 cache being core-specific, and subsequent levels being shared among cores. To address the issue further, specially in the case of vector calculations, manufacturers have also introduced SIMD (Single Instruction Multiple Data) which enables $\frac{1}{2^n}$ instructions to be executed in parallel, where n is the number of bits in the vector, thus reducing the overhead of communicating that many instructions.

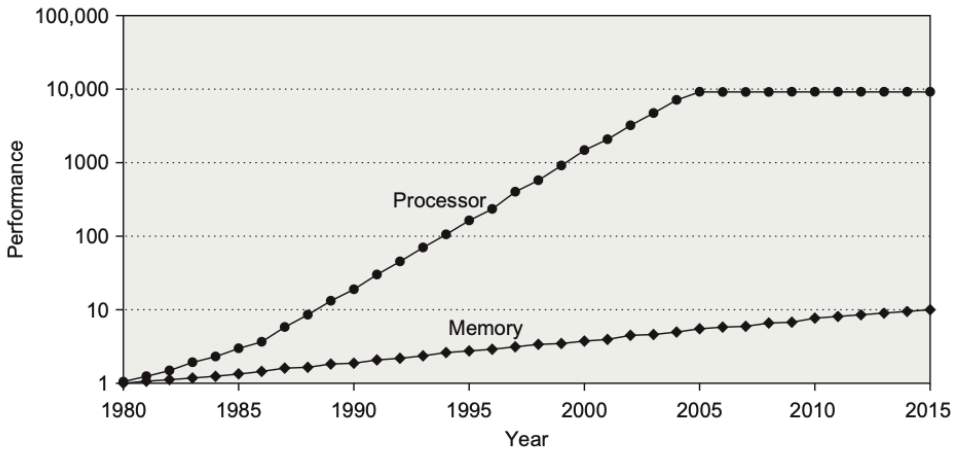

Figure 2: Difference in performance of processor and memory regarding the time difference for subsequent memory accesses in the processor versus the latency of DRAM access. Source: [4].

[3]This can be contrasted with *firmware*, which is code made for another processor, and is usually stored in rewritable memory. Firmware is needed to initialize and control hardware devices, like RAM controllers, or the Motherboard BIOS. This code can be written in C, but it is out of the scope of this work.

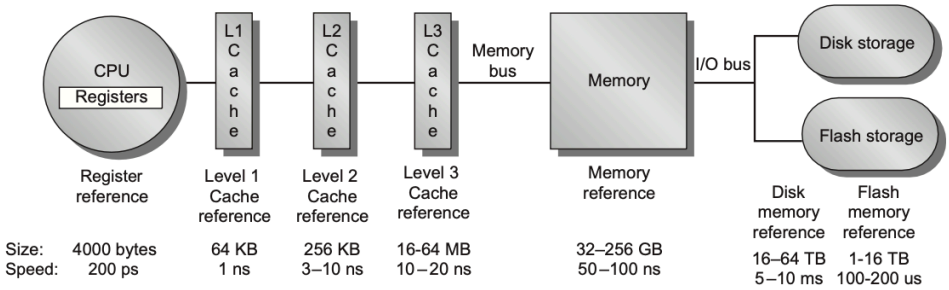

Figure 3: Memory hierarchy of a modern processor, showing the different levels of cache and their relation to the processor. Source: [4].

1.2. Operating Systems

Since operating system became very popular in the 90s, when computers where becoming commonplace, and the web emerged as a new medium, it is common to associate computers with their operating systems, making the case that they were a single whole, which is the reality for consumers. Historically, operating systems have been developed as a means of virtualizing physical resources, such as memory and compute, while also ensuring efficient operation of the hardware. This could be used for *time-sharing* systems, where many users could connect to the same computer and run their applications concurrently, without interfering with each other. R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau [5] describes three main aspects of an operating system:

- Virtualization

  The OS must provide interfaces for interacting with different hardware components, such as memory, storage, network cards, and peripherals. It also has to schedule execution of programs, deciding what to run next.

- Concurrency

  The OS has to provide concurrency primitives, such as mutexes and semaphores, to allow programs to make use of multi-core systems. It also has to enable threading and processes, however some operating systems provide different abstractions.

- Persistence

  The OS has to provide a way to store data, and to retrieve it. This can be done through filesystems, which are usually implemented as a tree of directories, with files at the leaves. The OS also has to provide a way to interact with the filesystem, such as reading and writing files and directories, and managing permissions.

Currently, there are few operating systems that are widely used, and most of them are based on the Unix operating system, which makes use of a kernel to manage resources, and a shell to interact with the user. The kernel is the main subject matter of operating systems, as it has direct access to the hardware, and is responsible for executing programs. There are, however, hybrid approaches, such as microkernels, which provides lower-level interfaces for user-space services to implement, relying on small programs to perform basic system tasks [1].[4]

Note that kernel-space and user-space are mostly divided by the instructions they can execute on CPU. If a program tries to execute an instruction that is not allowed, the kernel will raise an exception, and the program will be killed. The program must use the kernel interface to be able to ask for the physical resource, and can be denied depending on the running user's privileges.

[4]While it may seem unintuitive that the kernel should not be responsible for scheduling, B. P. Swift, T. Hardy, and A. Tanenbaum [6] highlights the main advantages of such decoupling.

We can already notice that the OS is very concerned with security, which comes at a big penalty for performance. For example, some recent hardware vulnerabilities have been patched by operating systems, such as Spectre and Meltdown, both related to out-of-order code execution. Performance issues with operating systems can also be traced to legacy interfaces to interact with hardware devices. In the case of filesystems and network cards, the overhead imposed by kernel interfaces can be up to 30% [7]. This can be mitigated with library operating systems, which bypass the kernel for specific tasks [8]. It is however, not in the scope of this work.

1.2.1. User-space dependencies

While most of what we discuss on this paper is related to end-user software libraries, it is important to note that the operating system is not only composed of kernel and shell on modern systems. There are many tools that aid in making an interactive experience, such as init systems, display managers, window managers, file managers and also many different libraries that are used by developers to make those tools, and which can be shared (dynamically linked) or statically linked (copied). All of those compete for CPU time, and even if not relevant to our analysis, they can be a source of performance issues, or just noise in the measurements. There are entire teams dedicated to understanding the impact of those services and tools in high-performance settings, and we take much inspiration from them in our analysis too.

1.3. State of user-space software development

With user-space software we refer to the software that we, as end-users, interact with. Most of this software is written in high-level languages, not only because they are less performance sensitive, but oftentimes because they target other software in which they will be run, e.g. a web browser or an isolated environment such as a container.[5] Most networked services have been design to scale horizontally, whereby the same application is run on multiple machines that are connected through a network, and can be added or removed as needed. This makes it easier for developers to maintain large applications, as they can be broken down into smaller parts that can each be scaled horizontally, in what is known as service-oriented architecture (SoA). SoA is highly encouraged in object-oriented programming, a mainstream paradigm promoted by influential authors and major companies, which developed many of the tools to build and manage microservices. This application oriented design is known for hiding details in favor of interfaces, and where most of the performance issues arise. It has become common in the industry to ignore the underlying layers of abstraction, and to focus on the end-user experience at the cost of several additional dependencies that snowball into performance issues. While it may seem like a good trade-off, it is not always the case, as oftentimes there is compounding additional cost to address the issue later.

The SoA approach comes in direct contrast to the lesser known data-oriented architecture (DoA), which we discuss in detail in the following section. It is important to note that no approach is a silver bullet, however data-oriented designs (DoD) strive to stay away from complexity while having proven their effectiveness[6], which can be very beneficial to modern machine-learning deployments, as well as scientific-computing as a whole, having the added advantage that its core tenets can be stated as guidelines for engineers and researchers working in those applications [9].

Contemporary science's reliance on software is a relatively new phenomenon. As such, research software is commonly not held at standards as high as more traditional research methods [10]. It is often developed by researchers inexperienced with real world development practices and, long-term sustainability is compromised [11].

[5] A container is a Linux-specific feature that enables applications to run sandboxed but use the same kernel as the host machine. It has become the most common way to deploy web applications, usually through Kubernetes or some other orchestration tool that enables horizontal scaling on commodity hardware.

[6] Especially in the games industry, where performance is paramount, and the software must run on a wide range of devices, from consoles to high-end PCs.

One particular development strategy that appeals to modern scientific standards is that of open source, in which the code is available to users. Notably, as open-source software is auditable, it becomes easier to verify reproducibility [12]. This also allows for early collaboration between researchers and developers, which can lead to better software design and performance [13].

Building on the practice of open source we also have *free software* (commonly denoted by FLOSS or FOSS): a development ideology centered on volunteer work and donations, and that is permissively (*copyleft*) licensed. There is emerging work on the role of FLOSS in science, such as L. Fortunato and M. Galassi [14], and some initiatives which praise a similar approach [15], [16]. We believe that the future of scientific computing lies in the hands of open-source communities that adopt data-oriented design principles, and that are able to leverage the latest hardware advancements.

In a field such as machine learning, it is common practice to implement and prototype algorithms in high-level languages such as Python, due to their ease of use, powerful features, and large ecosystem. Python is usually run by an interpreter, called CPython, which is written in C, and is responsible for translating it into machine code. An interpreter is a program that reads the source code of another program, translates it into bytecode, that is then interpreted by a virtual machine, which simulates a computer that understands that binary stream (as opposed to the actual host machine assembly, e.g. x86_64). Compare this with a compiler, which directly translates the source code into an executable binary. Python can also be compiled, at the cost of some flexibility, but it is also not as fast as systems programming languages.

It is not uncommon, however, to find that the performance of the code is not up to par with the requirements of running it for production, and it may be necessary to rewrite the application. A complete rewrite in a compiled language with manual memory management can be very time-consuming and error-prone. The most common alternative is to use libraries that provide bindings to compiled code, such as NumPy and Scikit-learn. Those bindings are called *foreign function interfaces* (FFI), and are used to call functions from one language in another, however most commonly they are used to call C functions.[7] We will now introduce some of the systems programming language alternatives we picked for analyzing in this research project.

1.3.1. Systems programming languages

C is the main example of a systems programming language. It has been successfully used to write such applications since its inception, as it was designed by Denis Ritchie to write the Unix operating system. Other languages are called systems programming languages because they are capable of delivering similar results to C, or because they could be used to eventually replace C, to write a kernel for example. The C language has become known for the wide range of vulnerabilities that can be exploited through *Undefined Behavior* (UB). There have been many attempts at language paradigms or compiler technology to provide an easy way to make safe code that is fast, and we will explore some of them. While old-standing competitors such as C++ have tried to address some issues with C, they have also become very complex, with their own set of pitfalls. Others, like Java, have tried to address the issue of memory management by providing a garbage collector, which tracks live objects by means of reference counting, and frees the memory when it is no longer needed. This adds overhead to the program, as the garbage collector must run periodically, and can be a source of performance issues, as the cost of garbage collection is not always predictable.[8]

A modern contender for systems programming is Rust, which promises on being a safe and performant alternative to C and C++. The Rust programming language has invested heavily in its type system, which can prevent many common programming errors by providing restrictive interfaces to the developer. Traditionally, an interface is a software contract that specifies some application boundary,

[7] Even though we will be exploring alternatives to C, it is important to note that all languages use the C binary calling convention to address foreign functions, even if they have a different ABI.

[8] A commonly known issue with garbage collectors is the *stop-the-world* problem, where the program must halt execution in order to perform garbage collection.

usually in the form of function signatures – the inputs and outputs of a (named) function. In Rust, those interfaces are called `traits`, which have an added semantic meaning of capabilities. A common example is serialization, whereby one converts the representation of an object. A Rust type that implements the `Serialize` trait can be converted to a JSON string, for example. These constraints are enforced by the compiler, and allow Rust to resolve symbols more efficiently than C++, being object-oriented, where it is common to have dynamic dispatch.[9] The main feature of Rust is, however, the borrow checker – this is a static analysis tool that builds on top of the type system, analyzing the lifetime of references to objects – which can prevent many common programming errors, such as *use-after-free* and *double-free*, freeing Rust code of major sources of undefined behavior. Prior to modern Rust, it was common to need explicit lifetimes in variables, but since that has become better integrated through compiler inference, the language has gained wide adoption for its advanced high-level features, usually called *zero-cost abstractions*.

Zig is another language that has gained some traction in the systems programming community. Through the use of `comptime` blocks, developers can achieve clean and readable code generation. Contrast it with the Rust alternatives of `macro_rules!` (which defines its own syntax) or AST-parsing macro functions which require in-depth knowledge of compiler internals. Zig also aims to be closer to the C language, promising on easy interoperability, and providing many ways to interface with C libraries, which are still very common in the systems programming community. The main selling point of Zig has been its simplicity, inspiretou by Go – a very simple interpreted language that has become quite popular for web development as it provides an easy way to write concurrent code, making it very maintainable and performant. While Zig still does not have concurrency primitives on par with Golang, another selling point is the ease of using allocators. When managing memory, allocators define policies to how it should be done, potentially reducing fragmentation and other latency issues related to acquiring and releasing resources from the Operating System. Traditionally, C and C++ have used the `malloc` and `free` functions, which are very simple, but can be a major source of issues, as the developer must remember to free their memory appropriately, and only once. Zig provides many different allocators that can be used, and also allows defining custom allocators, which are the only way to allocate memory in the language.

2. Background

2.1. First term

In October 2022 I have attended X.Org Developers Conference 2022 in order to present another project I have developed in Google Summer of Code 2022. While it was unrelated, I got to meet many people from the open-source graphics community, and learn much about the development of state of the art open source graphics APIs.

In the MAC0414 course, I have learned many things about the theory behind compilers, mainly through [17], which has helped me understand many ideas behind the compilation process.

Then, in the MAC0344 course I have learned about the main aspects guiding performance in modern hardware, and also about some of their most common optimizations.

2.2. Second term

I have learned about the main aspects of operating systems in the MAC5753 course, including various scheduling algorithms, details about memory management, as well as the main aspects of file systems design. This knowledge has already helped me break down multiple applications and systems I have encountered. This course was also very code-intensive, which has helped me improve my C program-

[9] Rust also supports dynamic dispatch through `dyn`, but heavily discourages its use.

ming skills. As part of the courses' evaluation, I have presented the Contiki-NG operating system for *Internet of Things* (IoT) applications.

In the MAC5742 course, I have learned about the main aspects of parallel programming, and studied code profiling and benchmarking. I have also made a group presentation about federated learning, which is a distributed machine learning technique.

Finally, I presented an extended abstract of my previous work at the 2023 ERAD-SP conference.

2.3. Third term

Both the MAC5716 and MAC5856 courses were focused on software development, and have helped me both to strengthen my engineering skills, and to learn about the intricacies of open source software development.

2.3.1. MAC5716 - Extreme Programming Laboratory

On this course, I learned about applying Agile methods to software development by contributing to SuperLesson, a lecture transcription CLI tool. Unlike most other projects in the course, our team already had, as a starting point, simple scripts that the client had started developing by himself. The scripts were a series of six steps, processing a video lecture and the professor's presentation slides, then producing a PDF file and the lecture transcription. The client then used to annotate each slide manually with the relative transcription, before using it for his studies.

To tackle this problem, we started by refactoring the code, so that we would understand every intricacy of the process, while also enabling us to add new features more easily, and convert the software into a CLI tool. We believe this choice was essential for making the software more maintainable, and helped us advance faster. After that, we started working on features that the client requested, to improve on the readability of the final output, and automate the process of annotating the slides. We have also added simple documentation, and linting tools to the project. Apart from new features, we have also shortened processing time in many steps. First, we made use of GPUs to generate the transcription from the lecture audio, which was the most time-consuming step, cutting its runtime by a factor of 20x. We were also able to improve both performance and accuracy of word splitting on slide transitions by tracking auto-generated punctuation. As a final improvement, we parallelized API calls to OpenAI's ChatGPT, which was the second most time-consuming step, cutting its runtime by the number of slides in the presentation.

Apart from developing software, we gathered every week to discuss our progress, and also with the client, on a separate occasion, for planning meetings. Our progress was tracked mainly through high-level tasks on a Kanban board, using GitHub Projects. This choice was made in favor of responsiveness and flexibility, as more detailed tracking would take too much time. We mainly worked through pair programming and pull requests during the course, but we also stopped reviewing pull requests to facilitate fast progress before delivery, with minor consequences on stability, as each change was tested by both developers. Our team worked on the project from september to december 2023.

2.3.2. MAC5856 - Open Source Software Development Laboratory

On this course, I learned about the various intricacies that are present on FOSS communities, and I have also contributed to a distributed version control system (DVCS) called Jujutsu (jj). Jj is a DVCS written in Rust, which aims to be a faster and more secure alternative to Git, and is currently still in its beta stage. The project was created by Martin von Zweigbergk and attempts to solve many problems that Git and other DVCSs have, mainly focusing on easy-of-use and performance.

During the course, my pair and I had to make presentations about code style, free software groups on the Southern Hemisphere, and the software project itself. Preparing and presenting the presentations helped us understand the topics more deeply, especially with respect to the software project, as we had to

Paper	Benchmark methods	OSS	Source
[18] J. Rotter and M. C. Lewis, "N-body performance with a kd-tree: Comparing rust to other languages," in *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2022, pp. 457–462.	Execution time Resident memory usage	Yes	github.com/MarkCLewis/MultiLanguageKDTree
[19] Y. Lin, S. M. Blackburn, A. L. Hosking, and M. Norrish, "Rust as a language for high performance GC implementation," *ACM SIGPLAN Notices*, vol. 51, no. 11, pp. 89–98, 2016.	Microbenchmarks Parallel performance scaling Domain-specific benchmarking tool (compares time against state-of-the-art implementation)	No	
[20] E. Schubert and L. Lenssen, "Fast k-medoids Clustering in Rust and Python," *Journal of Open Source Software*, vol. 7, no. 75, p. 4183, 2022.	MNIST sample runtime for a few values of N with fixed parameters	Yes	github.com/kno10/rust-kmedoids

Table 1: Research papers reviewed in this work.

look up many sources to motivate its creation, and also to understand how it differed from alternatives. This also allowed me to look more deeply into software design, and explore how time affect it.

3. Literature Review

Table 1 summarizes the main papers that have been reviewed in this work. Most modern authors do not spend much time discussing implementation details, with [18] being an exceptional example in many aspects.

4. Proposal

One of our goals in this work is to uncover how feasible it is to work consistently in a hybrid-language workflow, enabling high-performance Python through foreign function interfaces (FFI) to systems programming languages. We chose to investigate the performance of modern systems programming languages, namely Rust and Zig, as alternatives to C, in the context of high-performance scientific computing. For this purpose, we will be comparing the Python, C, Rust, and Zig implementations of a modern numerical algorithm that will be used in Python.

In order to compare Rust and Zig implementations to C, we hope to divide our analysis into three main aspects:

- Performance

  How fast the code runs, and how much memory does it use?

- Complexity

  How easy is it to understand and maintain the code?

- Maintainability

Implementation	LoC	avg timings (s)		
		avg ages	avg payments	stddev
Python	23	0.1907	1.9303	39.7549
NumPy	13	0.0371	0.1154	1.1720
C	138[10]	0.1933	1.9290	1.9450
Rust	40	0.0042	0.0362	0.5567
Zig	110[11]	0.2566	2.6005	2.6351

Table 2: Benchmark results for the toy statistics functions implementation.

How easy is it to extend and modify the code, in relation to its application?

As a motivating example for our work, we will be looking at a toy implementation of statistics functions across our languages of interest, with listings provided on Appendix A. In that example, we can see that the Rust implementation is very similar to the Python implementation, but can be faster than NumPy even if it is not loading the data with helper functions. For our first analysis, we will be using the Python implementation as a baseline, benchmarked using `timeit` with randomly generated data from 1000000 payment samples from 100000 users. The calculations are run 1000 times in a single Python interpreter session, for each implementation, displayed at Table 2.

We also point the reader to take a look at the Rust implementation in particular Listing 3, as it the fastest, while also being very similar to the pure Python version Listing 1 with a minor discrepancy in LoC (*lines of code*).

We present a timeline of the project in Figure 4, starting with the relevant studies the primary author began in 2022.

[10]The C implementation was written by the author, without any prior experience dealing with the Python extension API. It is still being reviewed by the author, and is expected to be improved.

[11]This implementation also made use of C headers directly, making it as complex as the C implementation.

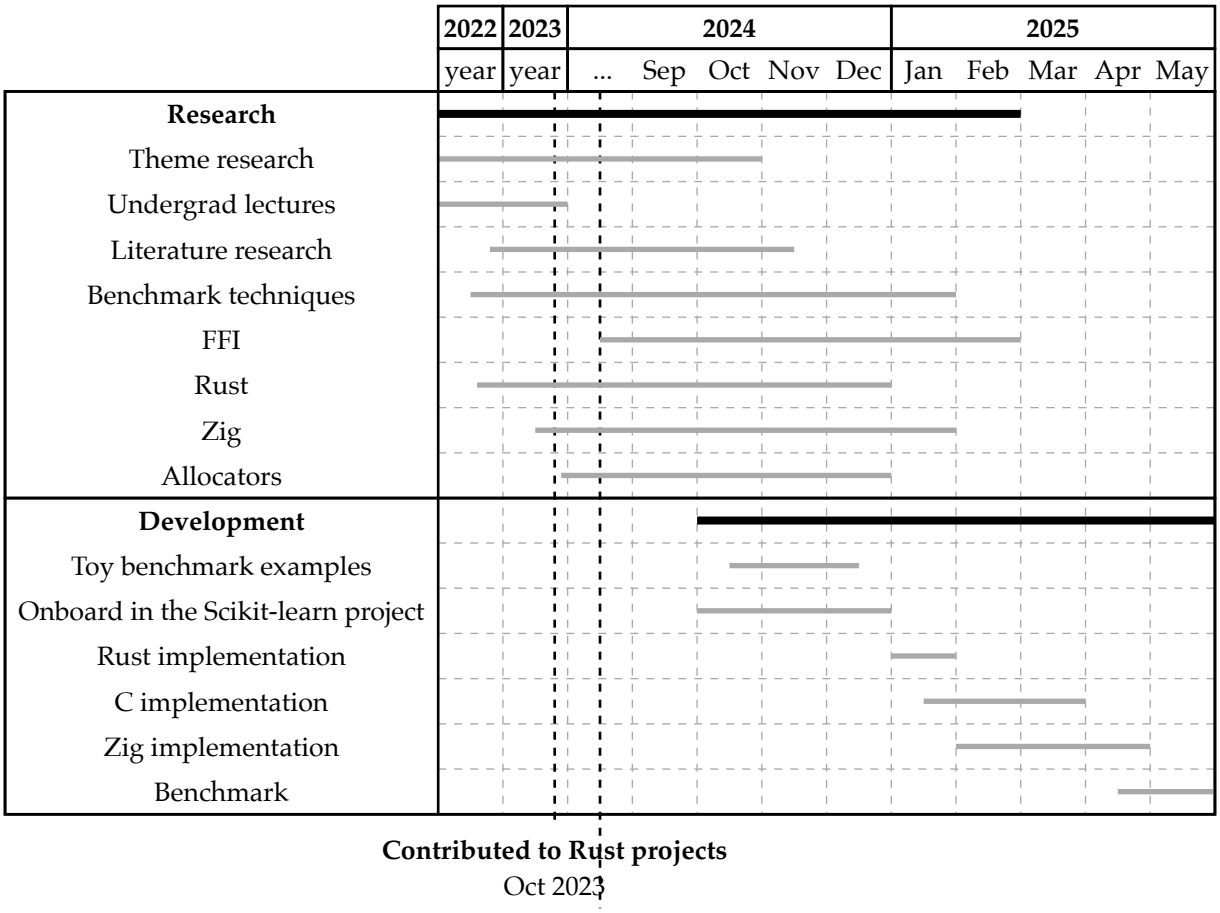

Figure 4: Timeline of the project.

A. FFI implementation reference

We have the following Python code to calculate statistics on a list of user data. Note that we prefer to use a single class with lists in order to pack our data as efficiently as possible, and we make use of `typing.NamedTuple` as a representative performant modern Python feature.

The NumPy implementation is as follows:

While using Rust with PyO3, we have the following implementation:

Due to time and resource constraints, we were still unable to determine enough API details through the Python external API to match Rust performance with a pure C, or a Zig implementation.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <math.h>
#include <stdlib.h>

typedef struct {
    PyObject_HEAD PyObject *ages;
    PyObject *payments;
} UserData;

static void UserData_dealloc(UserData *self);
static int UserData_init(UserData *self, PyObject *args, PyObject *kwds);
static PyObject *UserData_new(PyTypeObject *type, PyObject *args,
                              PyObject *kwds);
```

```
                    import math
                    import typing as t


                    class UserData(t.NamedTuple):
                        ages: list[int]
                        payments: list[int]

                        def average_age(self):
                            total_age = sum(self.ages)
                            count = len(self.ages)
                            return total_age / count

                        def average_payment_amount(self):
                            total_payment_cents = sum(self.payments)
                            count = len(self.payments)
                            return 0.01 * total_payment_cents / count

                        # Compute the standard deviation of payment amounts
                        # Variance[X] = E[X^2] - E[X]^2
                        def std_dev_payment_amount(self):
                            sum_square, total_sum = 0.0, 0.0
                            for payment_cents in self.payments:
                                payment = payment_cents * 0.01
                                sum_square += payment**2
                                total_sum += payment
                            count = len(self.payments)
                            avg_square = sum_square / count
                            avg = total_sum / count
                            return math.sqrt(avg_square - avg**2)
```

Listing 1: Pure Python implementation of the statistics functions.

```
static PyObject *UserData_average_age(UserData *self) {
    if (!PyList_Check(self->ages)) {
        return NULL;
    }

    Py_ssize_t len = PyList_Size(self->ages);
    if (len == 0) {
        return NULL;
    }

    double sum = 0.0;
    for (Py_ssize_t i = 0; i < len; i++) {
        PyObject *item = PyList_GetItem(self->ages, i);
        if (!PyLong_Check(item)) {
            return NULL;
        }
        sum += PyLong_AsLong(item);
    }

    return PyFloat_FromDouble(sum / len);
}

static PyObject *UserData_average_payment_amount(UserData *self) {
    if (!PyList_Check(self->payments)) {
        return NULL;
    }

    Py_ssize_t len = PyList_Size(self->payments);
    if (len == 0) {
```

```python
import typing as t

import numpy as np
import numpy.typing as npt


class UserData(t.NamedTuple):
    ages: list[int]
    payments: list[int]

    def average_age(self):
        total_age = sum(self.ages)
        count = len(self.ages)
        return total_age / count

    def average_payment_amount(self):
        total_payment_cents = sum(self.payments)
        count = len(self.payments)
        return 0.01 * total_payment_cents / count

    # Compute the standard deviation of payment amounts
    # Variance[X] = E[X^2] - E[X]^2
    def std_dev_payment_amount(self):
        sum_square, total_sum = 0.0, 0.0
        for payment_cents in self.payments:
            payment = payment_cents * 0.01
            sum_square += payment**2
            total_sum += payment
        count = len(self.payments)
        avg_square = sum_square / count
        avg = total_sum / count
        return math.sqrt(avg_square - avg**2)
```

Listing 2: (Python with) NumPy implementation of the statistics functions.

```c
        return NULL;
    }

    double sum = 0.0;
    for (Py_ssize_t i = 0; i < len; i++) {
        PyObject *item = PyList_GetItem(self->payments, i);
        if (!PyLong_Check(item)) {
            return NULL;
        }
        sum += PyLong_AsLong(item);
    }

    return PyFloat_FromDouble((sum * 0.01) / len);
}

static PyObject *UserData_std_dev_payment_amount(UserData *self) {
    if (!PyList_Check(self->payments)) {
        return NULL;
    }

    Py_ssize_t len = PyList_Size(self->payments);
    if (len == 0) {
        return NULL;
    }

    double sum_square = 0.0;
    double sum = 0.0;
```

```rust
use pyo3::prelude::*;

#[pymodule]
fn metrics_rs(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_class::<UserData>()
}

#[pyclass]
struct UserData {
    ages: Vec<u8>,
    payments: Vec<u32>,
}

#[pymethods]
impl UserData {
    #[new]
    fn new(ages: Vec<u8>, payments: Vec<u32>) -> Self {
        UserData { ages, payments }
    }

    /// Compute the average age
    fn average_age(&self) -> f64 {
        let sum: u64 = self.ages.iter().map(|&age| age as u64).sum();
        let count = self.ages.len() as f64;
        sum as f64 / count
    }

    /// Compute the average payment amount
    fn average_payment_amount(&self) -> f64 {
        let sum: u64 = self.payments.iter().map(|&p| p as u64).sum();
        let count = self.payments.len() as f64;
        (sum as f64 * 0.01) / count
    }

    /// Compute the standard deviation of payment amounts
    fn std_dev_payment_amount(&self) -> f64 {
        let mut sum_square = 0.0;
        let mut sum = 0.0;

        for &p in &self.payments {
            let x = p as f64 * 0.01;
            sum_square += x * x;
            sum += x;
        }

        let count = self.payments.len() as f64;
        let avg_square = sum_square / count;
        let avg = sum / count;

        (avg_square - avg * avg).sqrt()
    }
}
```

Listing 3: Rust PyO3 implementation of the statistics functions.

```c
for (Py_ssize_t i = 0; i < len; i++) {
    PyObject *item = PyList_GetItem(self->payments, i);
    if (!PyLong_Check(item)) {
        return NULL;
    }
    double payment = PyLong_AsLong(item) * 0.01;
    sum_square += payment * payment;
    sum += payment;
```

```
    }

    double avg_square = sum_square / len;
    double avg = sum / len;

    return PyFloat_FromDouble(sqrt(avg_square - avg * avg));
}

static PyMethodDef UserData_methods[] = {
    {"average_age", (PyCFunction)UserData_average_age, METH_NOARGS,
     "Compute average age"},
    {"average_payment_amount", (PyCFunction)UserData_average_payment_amount,
     METH_NOARGS, "Compute average payment amount"},
    {"std_dev_payment_amount", (PyCFunction)UserData_std_dev_payment_amount,
     METH_NOARGS, "Compute standard deviation of payment amounts"},
    {NULL} // Sentinel
};

static PyTypeObject UserDataType = {
    PyVarObject_HEAD_INIT(NULL, 0).tp_name = "metrics.UserData",
    .tp_doc = "UserData class",
    .tp_basicsize = sizeof(UserData),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = UserData_new,
    .tp_init = (initproc)UserData_init,
    .tp_dealloc = (destructor)UserData_dealloc,
    .tp_methods = UserData_methods,
};

static PyObject *UserData_new(PyTypeObject *type, PyObject *args,
                              PyObject *kwds) {
    UserData *self = (UserData *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->ages = PyList_New(0);
        self->payments = PyList_New(0);
        if (self->ages == NULL || self->payments == NULL) {
            Py_XDECREF(self->ages);
            Py_XDECREF(self->payments);
            Py_TYPE(self)->tp_free((PyObject *)self);
            return NULL;
        }
    }
    return (PyObject *)self;
}
```

```
const std = @import("std");

const py = @cImport({
    @cDefine("Py_LIMITED_API", "3");
    @cDefine("PY_SSIZE_T_CLEAN", {});
    @cInclude("Python.h");
});

pub export fn PyInit_metrics_zig() ?*py.PyObject {
    return py.PyModule_Create(&ModuleDef);
}

var ModuleDef = py.PyModuleDef{
    .m_base = .{
        .ob_base = .{
            .ob_type = null,
        },
        .m_init = null,
```

```c
static int UserData_init(UserData *self, PyObject *args, PyObject *kwds) {
    PyObject *ages = NULL, *payments = NULL;

    if (!PyArg_ParseTuple(args, "OO", &ages, &payments)) {
        return -1;
    }

    if (!PyList_Check(ages) || !PyList_Check(payments)) {
        return -1;
    }

    Py_INCREF(ages);
    Py_INCREF(payments);
    Py_XDECREF(self->ages);
    Py_XDECREF(self->payments);
    self->ages = ages;
    self->payments = payments;

    return 0;
}

static void UserData_dealloc(UserData *self) {
    Py_XDECREF(self->ages);
    Py_XDECREF(self->payments);
    Py_TYPE(self)->tp_free((PyObject *)self);
}

static PyModuleDef metricsmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "metrics_c",
    .m_doc = "Metrics module",
    .m_size = -1,
};

PyMODINIT_FUNC PyInit_metrics_c(void) {
    PyObject *m;
    m = PyModule_Create(&metricsmodule);
    PyModule_AddType(m, &UserDataType);
    return m;
}
```

Listing 4: C Python extension implementation of the statistics functions.

```
        .m_index = 0,
        .m_copy = null,
    },
    .m_name = "metrics_zig",
    .m_doc = "Metrics module implemented in Zig.",
    .m_size = -1,
    .m_methods = methods[0..],
    .m_slots = null,
    .m_traverse = null,
    .m_clear = null,
    .m_free = null,
};

var methods = [_]py.PyMethodDef{
    .{
        .ml_name = "average_age",
        .ml_meth = averageAgeCallback,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "Calculate the average age.",
    },
```

```
    .{
        .ml_name = "average_payment_amount",
        .ml_meth = averagePaymentAmountCallback,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "Calculate the average payment amount.",
    },
    .{
        .ml_name = "std_dev_payment_amount",
        .ml_meth = stdDevPaymentAmountCallback,
        .ml_flags = py.METH_VARARGS,
        .ml_doc = "Calculate the standard deviation of payment amounts.",
    },
};

fn averageAgeCallback(self: ?*py.PyObject, args: ?*py.PyObject) callconv(.C) ?*py.PyObject {
    _ = self;
    if (py.PyTuple_Size(args) != 1) return null;

    const pylist = py.PyTuple_GetItem(args, 0);
    if (pylist == null or py.PyList_Check(pylist) == 0) return null;

    const list_len = py.PyList_Size(pylist);
    if (list_len < 0) return null;

    var sum: f64 = 0;
    var i: isize = 0;
    while (i < @as(isize, @intCast(list_len))) : (i += 1) {
        const item = py.PyList_GetItem(pylist, i);
        if (item == null or py.PyLong_Check(item) == 0) return null;

        const age = py.PyLong_AsLong(item);
        if (age < 0) return null;
        sum += @floatFromInt(age);
    }

    const avg = sum / @as(f64, @floatFromInt(list_len));
    return py.PyFloat_FromDouble(avg);
}

fn averagePaymentAmountCallback(self: ?*py.PyObject, args: ?*py.PyObject) callconv(.C) ?
*py.PyObject {
    _ = self;
    if (py.PyTuple_Size(args) != 1) return null;

    const pylist = py.PyTuple_GetItem(args, 0);
    if (pylist == null or py.PyList_Check(pylist) == 0) return null;

    const list_len = py.PyList_Size(pylist);
    if (list_len < 0) return null;

    var sum: f64 = 0;
    var i: isize = 0;
    while (i < @as(isize, @intCast(list_len))) : (i += 1) {
        const item = py.PyList_GetItem(pylist, i);
        if (item == null or py.PyLong_Check(item) == 0) return null;

        const payment = py.PyLong_AsLong(item);
        if (payment < 0) return null;
        sum += @floatFromInt(payment);
    }

    const avg = 0.01 * sum / @as(f64, @floatFromInt(list_len));
```

```
    fn stdDevPaymentAmountCallback(self: ?*py.PyObject, args: ?*py.PyObject) callconv(.C) ?
    *py.PyObject {
        _ = self;
        if (py.PyTuple_Size(args) != 1) return null;

        const pylist = py.PyTuple_GetItem(args, 0);
        if (pylist == null or py.PyList_Check(pylist) == 0) return null;

        const list_len = py.PyList_Size(pylist);
        if (list_len < 0) return null;

        var sum_square: f64 = 0.0;
        var sum: f64 = 0.0;
        var i: isize = 0;
        while (i < @as(isize, @intCast(list_len))) : (i += 1) {
            const item = py.PyList_GetItem(pylist, i);
            if (item == null or py.PyLong_Check(item) == 0) return null;

            const payment = py.PyLong_AsLong(item);
            if (payment < 0) return null;
            const payment_float = @as(f64, @floatFromInt(payment)) * 0.01;
            sum_square += payment_float * payment_float;
            sum += payment_float;
        }

        const count: f64 = @floatFromInt(list_len);
        const avg_square = sum_square / count;
        const avg = sum / count;
        const std_dev = std.math.sqrt(avg_square - avg * avg);
        return py.PyFloat_FromDouble(std_dev);
    }
```

Listing 5: Zig Python extension implementation of the statistics functions.

```
    return py.PyFloat_FromDouble(avg);
}
```

References

[1] A. S. Tanenbaum, A. S. Woodhull, and others, *Operating systems: design and implementation*, vol. 68. Prentice Hall Englewood Cliffs, 1997.

[2] Wikipedia, "Von Neumann architecture — Wikipedia, The Free Encyclopedia ." 2024.

[3] D. D. Chen and G.-J. Ahn, "Security analysis of x86 processor microcode," *Ariz. State Univ. Tempe AZ USA Tech. Rep*, pp. 1–18, 2014.

[4] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC Boston, 2018.

[6] B. P. Swift, T. Hardy, and A. Tanenbaum, "Individual Programming Assignment User Mode Scheduling in MINIX 3," 2010.

[7] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam, "I'm not dead yet! the role of the operating system in a kernel-bypass era," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 73–80.

[8] I. Zhang *et al.*, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 195–211.

[9] C. Cabrera, A. Paleyes, P. Thodoroff, and N. D. Lawrence, "Real-world machine learning systems: A survey from a data-oriented architecture perspective," *arXiv preprint arXiv:2302.04810*, 2023.

[10] S. Sufi *et al.*, "Software in reproducible research: advice and best practice collected from experiences at the collaborations workshop," in *Proceedings of the 1st ACM sigplan workshop on reproducible research methodologies and new publication models in computer engineering*, 2014, pp. 1–4.

[11] J. C. Carver, N. Weber, K. Ram, S. Gesing, and D. S. Katz, "A survey of the state of the practice for research software in the United States," *PeerJ Computer Science*, vol. 8, p. e963, 2022.

[12] L. A. Barba, "Defining the role of open source software in research reproducibility," *arXiv preprint arXiv:2204.12564*, 2022.

[13] G. Wilson *et al.*, "Best practices for scientific computing," *PLoS biology*, vol. 12, no. 1, p. e1001745, 2014.

[14] L. Fortunato and M. Galassi, "The case for free and open source software in research and scholarship," *Philosophical Transactions of the Royal Society A*, vol. 379, no. 2197, p. 20200079, 2021.

[15] D. S. Katz *et al.*, "Community organizations: Changing the culture in which research software is developed and sustained," *Computing in Science & Engineering*, vol. 21, no. 2, pp. 8–24, 2018.

[16] M. Barker *et al.*, "Introducing the FAIR Principles for research software," *Scientific Data*, vol. 9, no. 1, pp. 1–6, 2022.

[17] M. Sipser, "Introduction to the Theory of Computation," *ACM Sigact News*, vol. 27, no. 1, pp. 27–29, 1996.

[18] J. Rotter and M. C. Lewis, "N-body performance with a kd-tree: Comparing rust to other languages," in *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2022, pp. 457–462.

[19] Y. Lin, S. M. Blackburn, A. L. Hosking, and M. Norrish, "Rust as a language for high performance GC implementation," *ACM SIGPLAN Notices*, vol. 51, no. 11, pp. 89–98, 2016.

[20] E. Schubert and L. Lenssen, "Fast k-medoids Clustering in Rust and Python," *Journal of Open Source Software*, vol. 7, no. 75, p. 4183, 2022.