JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 35 (2025) 137–141
Received: 10 January 2025
Accepted: 11 February 2025

Comparative analysis of Python and Rust: evaluating their combined impact on performance

Analiza porównawcza Python oraz Rust: wpływ ich połączenia na wydajność

Przemysław Mroczek*, Jakub Mańturz, Marek Miłosz

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
This article presents results of exploration of the combined application of Python and Rust in enhancing software performance, focusing on algorithm implementation. Python, known for its simplicity and flexibility, is widely used in various domains but faces limitations in CPU-intensive tasks. Rust, on the other hand, excels in performance and safety of memory management, making it a compelling choice for optimizing critical code sections. By leveraging tools such as PyO3 and Maturin, Authors examine how Rust's compiled code can be seamlessly integrated into Python projects to mitigate performance bottlenecks.

Keywords: Python; Rust; Pyo3, Maturin

Streszczenie
W niniejszym artykule zaprezentowano wyniki badań wydajności połączenia technologii Python i Rust w celu poprawy wydajności oprogramowania, ze szczególnym uwzględnieniem implementacji algorytmów. Python, znany ze swojej prostoty i elastyczności, jest szeroko wykorzystywany w różnych dziedzinach, jednak napotyka ograniczenia w zadaniach intensywnie korzystających z procesora. Z kolei Rust wyróżnia się wydajnością i bezpiecznym zarządzaniem pamięcią, co czyni go atrakcyjnym wyborem do optymalizacji krytycznych fragmentów kodu. Wykorzystując narzędzia takie jak PyO3 i Maturin, poddano analizie, w jaki sposób skompilowany kod w języku Rust może być płynnie zintegrowany z projektami wykorzystującymi Python, aby ograniczyć elementy aplikacji negatywnie wpływające na wydajności.

Słowa kluczowe: Python; Rust; Pyo3, Maturin

*Corresponding author
Email address: przemyslaw.mroczek1@pollub.edu.pl (P. Mroczek)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

Python has consistently ranked among the most popular programming technologies in recent years [1]. Its popularity is due to the ease of writing code, readable syntax, and the ability to quickly prototype thanks to interpreter technology. The language has found wide applications in fields such as data science, machine learning, task automation, and web development [2, 3]. Its biggest drawback, however, is its relatively low efficiency compared to compiled technologies, which makes it not always suitable for tasks requiring maximum efficiency. Rust is a relatively new programming technology, the development of which began in 2006, and since 2009 the project has gained support from the Mozilla Foundation [4]. Its goal was to create a high-performance language that also provides memory safety and the ability to effectively manage parallelism. Thanks to these features, Rust has become an attractive alternative to languages such as C and C++, eliminating many typical errors related to memory management. Currently, Rust is great for applications such as backend development, game engines, blockchain systems, and performance optimization of applications written in other languages such as Python [5]. Although Python is versatile, its performance in CPU-intensive operations is limited. Rust offers the ability to write high-performance code fragments that can be seamlessly used as modules in Python applications.

With tools such as PyO3 and Maturin, developers can integrate Rust with Python, optimizing critical parts of the application while maintaining simplicity of programming and application security [6]. The main objective of research is to demonstrate the benefits of implementing core application functionalities using Rust and seamlessly integrating them with Python-based applications. To achieve this, Authors put forward the thesis: **Rust is more efficient in terms of time complexity than Python.** To prove this, the following hypotheses were defined:
H1: Algorithms implemented in Rust will be executed faster than algorithms implemented in Python.
H2: Algorithms running in the Docker environment will be executed faster than in the Windows environment.

2. Literature overview

Modern software development often benefits from combining interpreted and compiled languages. Python's flexibility can be enhanced with compiled code to offset performance limitations [7]. Rust has emerged as a viable option for this purpose, with tools like PyO3 and Maturin enabling the seamless integration of Rust code into Python projects [8]. This hybrid approach has been

demonstrated in projects such as Polars [9], which combines Rust's computational power with Python's user-friendly interface for data analysis, achieving significant performance gains. In modern Python programming, frameworks like FastAPI are gaining popularity for web development, heavily relying on Pydantic for data validation [10]. Pydantic enhances Python's type-checking capabilities, making it more akin to strongly typed languages [11]. Initially written in Python, Pydantic underwent a substantial performance transformation when its core was rewritten in Rust. The result, Pydantic V2, is 5–50 times faster than its predecessor [11]. Rust also supports tools that enhance the efficiency of Python codebases in enterprise settings.

Ruff, a lightweight linter for Python, is 10–100 times faster than traditional linters like Flake8 and linters like Black, making it a preferred choice for large-scale codebases [12].

UV, a package manager and dependency resolver written in Rust, is designed to replace tools like Pip, Pip-tools, and Virtualenv, offering improved speed and reliability in managing Python environments. UV achieves 8-10 times faster performance boost without caching [13, 14].

By integrating Rust-based solutions, Python developers can address performance bottlenecks while maintaining the language's inherent simplicity and versatility. This combination is increasingly vital in developing scalable and high-performing applications.

3. Objective and scope of the study

This research project aims to compare the time efficiency of algorithm executions, focusing on Bubble Sort, Insertion Sort, the Knapsack Problem (solved using a brute-force approach), and finding prime numbers using the Sieve of Eratosthenes [15, 16]. These algorithms will be implemented in both Python and Rust. Specifically, the study will analyze the performance of native Python implementations versus Rust implementations compiled and integrated for use in Python.

The objective is to explore potential performance improvements achievable through the integration of Rust's compiled code while leveraging Python's simplicity and ease of use. By conducting a detailed analysis of execution times and resource utilization, the study seeks to identify scenarios where Rust provides significant performance advantages. The findings will culminate in recommendations for optimizing algorithm implementations tailored to specific performance requirements, highlighting the trade-offs between native Python code and Python integrated with compiled Rust.

4. Research implementation

Considering the prerequisites for conducting the experiment correctly, we selected a laptop equipped with suitable components (Table 1).

Table 1: Test Environment Hardware

Processor	Intel Core i7-13700HX
SSD	SKHynix_HFS512GEJ9X115N
GPU	NVIDIA GeForce RTX 4070 Laptop GPU
RAM	DDR5-5600 / PC5-44800 DDR5 SDRAM SO-DIMM

The software used for implementing algorithms and conducting benchmarks is presented in Tab. 2.

Table 2: Test Environment Software

Python	Version 3.12.3
Python IDE	JetBrains PyCharm Professional 2024.1.3
Rust	Version 1.81.0
Rust IDE	JetBrains RustRover 2024.1.3

These tools and configurations ensure a standardized environment for executing and comparing algorithm performance between Python and Rust, providing reliable results for evaluation and analysis.

To conduct the study, the selected algorithms were implemented in Python (version 3.12.3) and Rust (version 1.81) [17, 18]. Both implementations strictly followed pseudocode-based logic, ensuring that no technology-specific optimizations were applied (Listing 1-2). This approach provides a fair comparison between the two languages.

Listing 1: Python implementation of Bubble Sort

```python
@timing_decorator
def bubble_sort_python_implementation(A):
    n = len(A)
    for i in range(n):
        swapped = False
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
                swapped = True
        if not swapped:
            break
    return A
```

Listing 2: Rust implementation of Bubble Sort

```rust
#[pyfunction]
fn bubble_sort(arr: Vec<i32>) -> PyResult<Vec<i32>> {
    let mut a = arr.clone();
    let n = a.len();
    for i in 0..n {
        let mut swapped = false;
        for j in 0..(n - i - 1) {
            if a[j] > a[j + 1] {
                a.swap(j, j + 1);
                swapped = true;
            }
        }
        if !swapped {
            break;
        }
    }
    Ok(a)
}
```

For both Rust and Python, the code was exported as Python-compatible libraries in the form of wheel files. These libraries were subsequently installed and tested within a Docker-based testing environment built on the python:3.12-slim image and were run manually in the PyCharm application on Windows environment. This containerized setup ensures a hermetic test environment, allowing for isolated and reproducible testing.

The use of compiled Python libraries (.whl files) for Rust implementations ensures a consistent testing framework across both languages. Execution times for each algorithm were measured using a Python decorator function specifically designed to capture execution time (Listing 3) and hermetic benchmark environments. This methodology guarantees unambiguous results, as both implementations are evaluated under identical conditions within the same environment.

Listing 3: Function measuring execution time

```
16 usages   pmroczek *
def timing_decorator(func):
    """
    Decorator to measure execution time of a function
    and log to a CSV file.
    """
    pmroczek *
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        execution_time = time.perf_counter() - start_time

        log_execution_time(func.__name__, execution_time)
        print(f"{func.__name__} executed in: "
              f"{execution_time:.9f} seconds")

        return result
```

The tests were conducted on a sample of 100 datasets, each consisting of 10,000 unsorted integers ranging from 1 to 100,000,000. For each dataset, the same set of test data was used for both the Python implementation and the Rust implementation. Each test run utilized a newly generated dataset and was executed within a freshly created Docker container (Listing 4). The only exception was the Sieve of Eratosthenes where the input data was an integer of 100,000,000.

Listing 4: Script running hermetic benchmark environments

```
#!/bin/bash

IMAGE_NAME="rust"
CONTAINER_NAME="rust"
NUM_RUNS=100

for i in $(seq 1 $NUM_RUNS); do
    echo "iteration: $i"
    echo "Building Docker image $IMAGE_NAME..."
    docker build -t $IMAGE_NAME .

    echo "Starting container $CONTAINER_NAME..."

    docker run -d -v $(pwd)/results:/app/results --name $CONTAINER_NAME $IMAGE_NAME

    sleep 3

    echo "Removing docker image and container"
    docker stop $CONTAINER_NAME
    docker rm $CONTAINER_NAME
    docker rmi $IMAGE_NAME
done
```

5. Results

The collected results include the average, minimum, and maximum times, along with the standard deviations. All the data are summarized in Tables 3-10, while the outcomes of each test probe are illustrated in the accompanying figures.

Table 3: Bubble Sort comparison, Docker environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	4,2131	3,6745	3,5053	0,1159
Rust	0,0461	0,0322	0,0288	0,0025

Table 4: Insertion Sort comparison, Docker environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	1,4469	1,2555	1,1825	0,0442
Rust	0,0192	0,0154	0,0143	0,0009

Table 5: Knapsack Problem comparison, Docker environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	26,5826	20,5754	18,8939	1,8440
Rust	0,2381	0,2099	0,1930	0,0094

Table 6: Sieve of Eratosthenes comparison, Docker environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	12,0812	11,0645	10,2454	0,3135
Rust	1,0053	0,9334	0,8854	0,0242

Table 7: Bubble Sort comparison, Windows environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	4,8812	4,6891	4,6269	0,0499
Rust	3,0535	2,9753	2,9434	0,0286

Table 8: Insertion Sort comparison, Windows environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	1,6542	1,5913	1,5659	0,0208
Rust	0,7423	0,7052	0,6933	0,0101

Table 9: Knapsack Problem comparison, Windows environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	0,5754	0,5395	0,5270	0,0086
Rust	0,2456	0,2135	0,2072	0,0054

Table 10: Sieve of Eratosthenes comparison, Windows environment

	max time (s)	avg time (s)	min time (s)	std.dev. (s)
Python	16,6701	15,6758	15,3628	0,2052
Rust	12,9289	12,4777	12,3336	0,0925

In Figures 1–8, the experiment results are presented in the form of times for each examined sample.

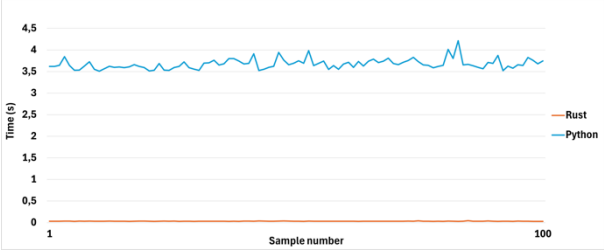

Figure 1: Bubble sort comparison, Docker environment

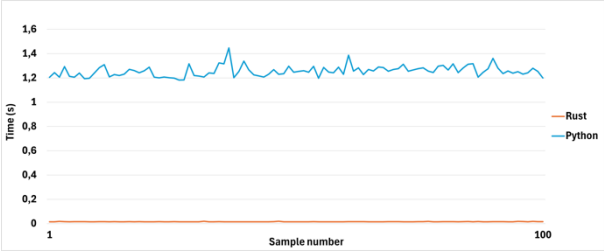

Figure 2: Insertion Sort comparison, Docker environment

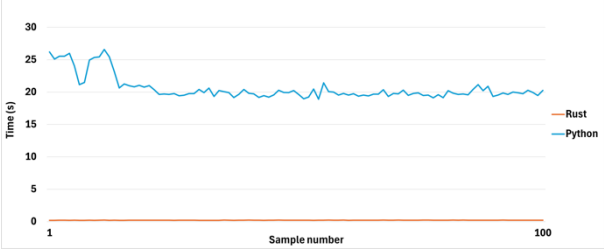

Figure 3: Knapsack problem comparison, Docker environment

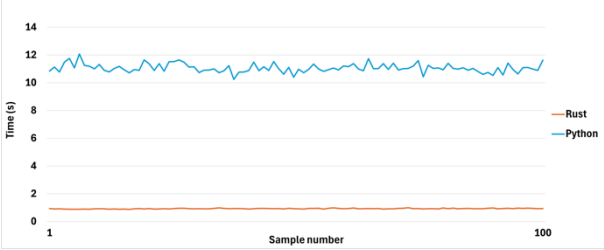

Figure 4: Sieve of Eratosthenes problem comparison, Docker environment

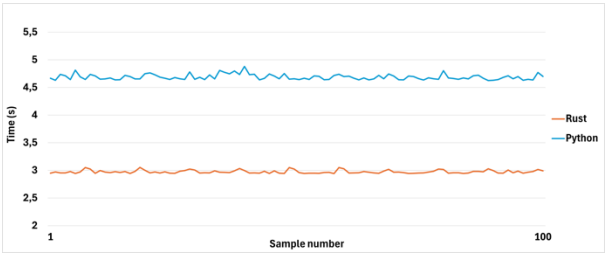

Figure 5: Bubble sort comparison, Windows environment

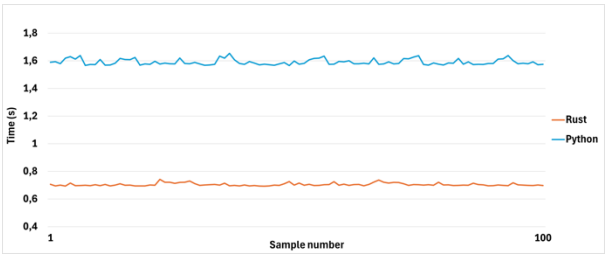

Figure 6: Insertion Sort comparison, Windows environment

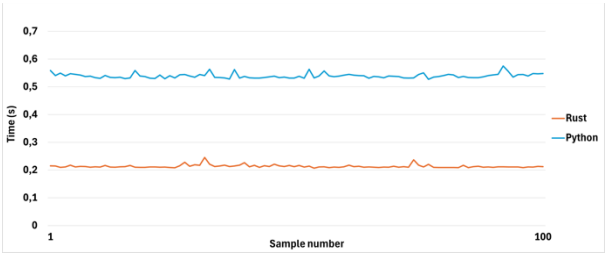

Figure 7: Knapsack problem comparison, Windows environment

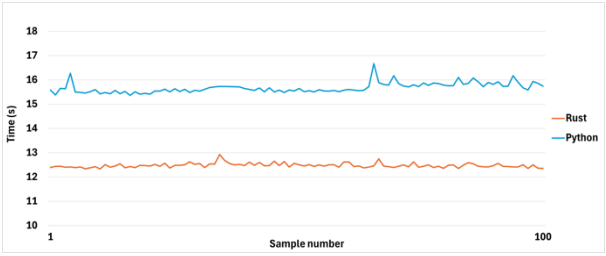

Figure 8: Sieve of Eratosthenes problem comparison, Windows environment

6. Discussion

The results of the experiments (see Fig. 1-8) clearly indicate that the Rust implementation significantly outperforms the Python implementation in terms of the amount of time the algorithms need to perform operations on the data with minimal costs associated with a slight increase in memory consumption.

A comparison of the research results conducted by Fanila Ali Agha and Haque Nawaz [15] with our results indicates Rust's dominance in performing low-level operations [15]. It should be noted that the Maturin and PyO3 libraries require at least Python 3.8 [19].

In applications using older versions of Python, it is impossible to convert Rust code into a Python module. Additionally, the specially prepared, isolated environment may not fully reflect real-world conditions.

Consequently, the final performance may differ from the results presented in our study.

The analysis of the data we collected clearly indicates the dominance of Rust over Python in projects requiring fast data processing. However, the simplicity of implementation and the speed at which the code is delivered make Python still a competitive solution.

In the continuation of our research, it would be worth considering the use of other languages such as C or C++ to collect research samples. Another possible idea to be implemented is the implementation of algorithms whose time and memory complexity would be greater than the algorithms used in these studies.

7. Conclusions

The analysis of the results obtained of the conducted research clearly indicates a significant reduction in the execution time of algorithms, which directly translates into the efficiency of the application. The only cost incurred to reduce the time needed to execute the algorithms is a slight increase in memory consumption.

The analysis of the research results allowed for the verification of the proposed hypotheses and thesis. Hypothesis H1 - **Algorithms implemented in Rust will be executed faster than algorithms implemented in Python** has been confirmed. As evidenced by the presented data (see Fig. 1–8 and Tab. 3–10), algorithms implemented in Rust require less time to process data compared to their Python counterparts. Hypothesis H2 - **Algorithms running in the Docker environment will be executed faster than in the Windows environment** has also been validated. Algorithms executed within the Docker environment completed their tasks significantly faster than those run in the Windows environment (see Fig. 1-8 and Tab. 3-10). Considering the positive confirmation of both hypotheses, it can be unequivocally stated that the thesis: **Rust is more efficient in terms of time complexity than Python** has been substantiated. What is more, the library containing the implementation of Rust algorithms only slightly contributes to the memory consumption of the application.

References

[1] Tiobe index, https://www.tiobe.com/tiobe-index/, [21.11.2024].

[2] Stack Overflow Annual Developer Survey, https://survey.stackoverflow.co/, [21.11.2024].

[3] T. E. Oliphant, Python for Scientific Computing, Computing in Science & Engineering 9(3) (2007) 10–20, https://doi.org/10.1109/MCSE.2007.58.

[4] A. Alheraki, The Journey of Rust: From Individual Effort to Mozilla's Backing, https://www.linkedin.com/pulse/journey-rust-from-individual-effort-mozillas-backing-ayman-alheraki-g0psf/, [05.01.2025].

[5] J. M. Perkel, Why scientists are turning to Rust, Nature 588 (2020) 185–186, https://doi.org/10.1038/d41586-020-03382-2.

[6] PyO3 technical documentation, https://pyo3.rs/v0.23.1/, [21.11.2024].

[7] M. Flitton, Speed Up Your Python with Rust, Packt Publishing Ltd, Birmingham, 2022.

[8] H. Lunnikivi, K. Jylkkä, T. Hämäläinen, Transpiling Python to Rust for Optimized Performance, Proceedings of the 20th International Conference Embedded Computer Systems: Architectures, Modeling, and Simulation (2020) 127–138, https://doi.org/10.1007/978-3-030-60939-9_9.

[9] Polars technical documentation, https://pola.rs/, [05.01.2025].

[10] FastAPI technical documentation, Pydantic features, https://fastapi.tiangolo.com/features/#pydantic-features, [05.01.2025].

[11] Pydantic 1 vs 2: A Speed Comparison, https://janhendrikewers.uk/pydantic-1-vs-2-a-benchmark-test, [21.11.2024].

[12] Ruff Overview, https://docs.astral.sh/ruff/, [03.12.2024].

[13] S. Gawande, Uv - pip killer or yet another package manager?, https://blog.kusho.ai/uv-pip-killer-or-yet-another-package-manager/, [03.12.2024].

[14] S. Katabatunni, UV vs. PIP: Revolutionizing Python Package Management, https://medium.com/@sumakbn/uv-vs-pip-revolutionizing-python-package-management-576915e90f7e, [03.12.2024].

[15] F. A. Agha, H. Nawaz, Comparison of Bubble and Insertion Sort in Rust and Python Language, IJATCSE, 10(2) (2021) 1020–1025, https://doi.org/10.30534/ijatcse/2021/761022021.

[16] H. A. Helfgott, An improved sieve of Eratosthenes, Math. Comp 89 (2020) 333–350, https://doi.org/10.1090/mcom/3438.

[17] Python3.12 technical documentation, https://docs.python.org/3.12/, [21.11.2024].

[18] Rust technical documentation, https://prev.rust-lang.org/en-US/documentation.html, [21.11.2024].

[19] Maturin technical documentation, https://www.maturin.rs/, [06.01.2025].