Volume: 12 Issue: III Month of publication: March 2024
DOI: https://doi.org/10.22214/ijraset.2024.58903

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 12 Issue III Mar 2024- Available at www.ijraset.com

Reconsidering Python Syntax to Enhance Programming Productivity

Chengze Ye[1], Zhuoyang Shen[2], Yue Wu[3], Pavel Loskot[4]
ZJU-UIUC Institute, Zhejiang University

Abstract: Data analytics plays a crucial role in today's society across various domains, driven by technological advancements and exponential data growth. Handling large-scale data poses a challenge due to increased computational and storage requirements. The heterogeneity of tasks in data analytics programming languages complicates integration and interaction, necessitating effective cross-language integration for productivity and extended capabilities. This paper proposes a generalized interpreter accepting various language syntaxes, primarily based on Python and MATLAB, with comparisons to R and Julia. Findings reveal Python's beginner-friendly learning curve and rich resources, Julia's high-performance computing, MATLAB's numerical prowess and specialized toolbox, and Python and R's focus on flexibility. Both Python and R boast active communities, while Python offers extensive portability, and Julia emphasizes interoperability. Despite syntactic differences, a common interpreter offers flexibility and efficiency, benefiting developers by enabling language selection based on project needs. Challenges can be mitigated through good design and technical solutions. Encouragement for research and innovation in universal interpreter development fosters collaboration, enhancing opportunities in data analysis and scientific computing. Active participation from developers and researchers is encouraged for continual improvement and advancement in the field.
Keywords: programming languages, language syntaxes, common interpreter, Python, MATLAB

I. INTRODUCTION

A. Background

The importance and wide range of applications of data analytics in today's society and in various fields has become more and more evident. With the rapid development of technology and the explosion of data, data analytics has become a key tool for everything from scientific research to business decision making. In today's world of high data processing demands, the field of data analytics is faced with the challenge of handling large-scale data [1]. The size and complexity of large data sets may lead to increased computation and storage requirements, and traditional compilers for data processing programs may become inefficient or infeasible. The alienation of the tasks undertaken by the major data analytics programming languages is one of the reasons for the above problems. The field of data analysis often involves the use of multiple programming languages and tools. Integration and interaction between different languages can be challenging, including data format conversions, differences in syntax and semantics, compatibility of libraries and functions, and so on. Effective cross-language integration is key to improving productivity and expanding data analysis capabilities. And learning and adapting to new techniques and tools may require continuous learning and updating of knowledge, which can be challenging for practitioners.

B. Objective

In this situation, the main goal of this paper is to propose a universal interpreter that accepts syntax of different languages [2]. This multi-language interpreter is mainly based on the programming environment of Python and powerful algorithmic capabilities of MATLAB. However, considering that R and Julia languages also have some unique features in their algorithms, we will make a cross-sectional comparison and explore these four languages. We also explain why we choose to propose an interpreter that accepts different syntaxes based on several reasons such as learning curve, computational speed, and community support.

C. Previous Literature

In a study, Šipek et al. introduce us to GraalVM, a high-performance multilingual virtual machine that supports the simultaneous operation and interoperability of multiple programming languages [3]. GraalVM runs Java efficiently, and we can use multiple languages simultaneously in this virtual machine, such as JavaScript, R, etc. GraalVM's main performance advantage of GraalVM comes from its use of Graal compiler technology, which is a JIT (Just-In-Time) compiler that converts bytecode to machine code at runtime and dynamically optimizes code execution to improve program execution efficiency.

In summary, the emergence of GraalVM opens up new possibilities for multilingual development and interoperability. It provides ideas and technical basis for realizing universal interpreters that can accept different grammars. At the same time, we can consider utilizing high-performance compiler technology similar to GraalVM to improve the execution efficiency and performance of the interpreter. This project provides a blueprint for our exploration of the feasibility of a multilingual compiler with data analysis as a requirement.

D. Expected Results

Through the feasibility analysis of this study, we hope to provide people in the field of scientific computing and data analysis with a comprehensive view of the characteristics of these four languages based and evaluate the feasibility of proposing a universal interpreter that accepts syntax of different languages. This will provide users with a more flexible and economical choice of tools, facilitate knowledge exchange and technical cooperation among programming languages, compensating for the shortcomings that now arise from the independence of each language.

II. METHODOLOGY

A. Objective Setting

Based on the goal of this research: to reconsider the current dilemma of Python syntax in the field of data analytics and to enhance programming productivity, we had a thorough discussion and finally decided to propose a multilingual integrated compiler in the context of data analytics and to explore its feasibility in the context of current applications.

B. Literature Review

Based on our knowledge of the topic, we searched for relevant literature in the following directions: the background of the application of data analytics and its current dilemmas; information on the syntactic environment, learning curve, and community support of the languages (Python, MATLAB, R, Julia) that serve as the main tasks of data analytics; and information on previous multilingual compilers.

C. Programming Language Comparisons

After the Literature review, we compare and analyse four commonly used languages for data analysis. The comparison dimensions are Learning Curve, Speed of Computations, Community Support, Debugging Levels, Portability and Syntax.

D. Programmatic Statement

After comparing the languages, we propose some feasible programs and screen them. In this process, we emphasize the strengths and weaknesses of different programming languages and aim to develop the design in a way that maximizes the strengths and avoids the weaknesses. The selection of programs is based on the potential benefits, beneficiary groups, social impacts, etc. of different programs.

III.DISCUSSION AND ANALYSIS

A. Learning Curve

The learning curve is the curve in which the time and effort required for the learning process changes as experience is gained. A flat learning curve indicates relatively easy learning, and a steep one indicates relatively difficult learning.

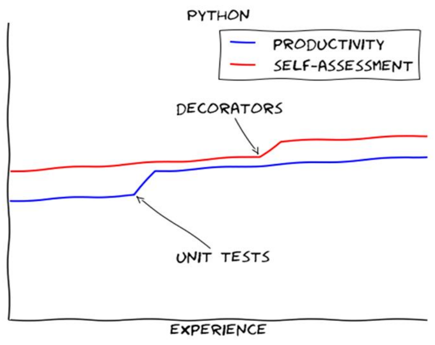

Fig. 1 Python learning curve [4]

Python's learning curve is relatively gentle, making it suitable for beginners to get started quickly. The simple syntax and abundant resources make the learning process relatively easy and enjoyable. Matlab has a steep learning curve and requires some time and effort to master its unique syntax and tools. Julia has a steeper learning curve, but it has an intuitive and consistent syntax design, as well as a wealth of learning resources and support. The R language is mainly used for statistical computation and graphing, and it has a relatively gentle learning curve for data analysis tasks. However, for routine programming tasks, R may not be as intuitive as other languages.

B. Speed of Computations

Different Python interpreters can have an impact on the speed of operation, e.g., CPython, PyPy, etc. Some data types (e.g., NumPy arrays) operate faster than others. Choosing efficient algorithms and data structures can increase the speed of operations. In addition, integration with GPUs can also speed up operations by utilizing their parallel computing power, especially in scientific computing and machine learning [5].

MATLAB uses a Just-In-Time Compiler to convert MATLAB code to machine code to improve performance. Different versions of MATLAB may have different compiler optimization strategies, which may affect computing speed. MATLAB provides many built-in functions and toolboxes, some of which are highly optimized to provide fast numerical calculations and matrix operations. Use of these built-in functions and toolboxes can result in high computational speed [6].

Julia uses Just-In-Time Compilation (JIT) technology to compile Julia code into native machine code in real-time for improved performance. This compilation allows Julia to dynamically optimize code at runtime and is often comparable to statically compiled languages. Julia supports multi-threading and parallel computing, allowing computational tasks to be parallelized to make better use of multi-core processors and cluster resources [7].

R has packages dedicated to high-performance computing, such as data, table and dplyr, which provide faster data manipulation. It also provides some parallel computing features, such as the parallel package and foreach package, which can be used to accelerate computation using multi-core processors or clusters. The R language also supports accelerating task-specific computation by compiling extension packages. For example, using the Rcpp package you can embed C++ code into R code to take advantage of the performance benefits of C++ [8].

In summary, Python has an advantage in its extensive third-party library and tool support, such as NumPy and Pandas, as well as its ability to integrate with other languages. Matlab has an advantage in its powerful numerical computation capabilities and toolkit for specialized domains. Julia has an advantage in performance optimization and was designed initially to provide high-performance computing. R has an advantage in statistical analysis and data processing with a rich set of statistical functions and extension packages.

However, Python is limited by Global Interpreter Lock (GIL) for multi-threaded parallel computation, Matlab has a high cost of commercial licensing, Julia has a relatively small ecosystem, and R may be weak in handling large-scale data and performance optimization. Thus, when choosing a language to fit a particular need, one needs to weigh their strengths and weaknesses and make decisions on a case-by-case basis.

C. Community Support

The data comparison of the most popular programming languages in 2022 had 71,547 people responding, with python at 48.07%, Matlab at 4.1%, Julia at 1.53%, and R at 4.66% [9].

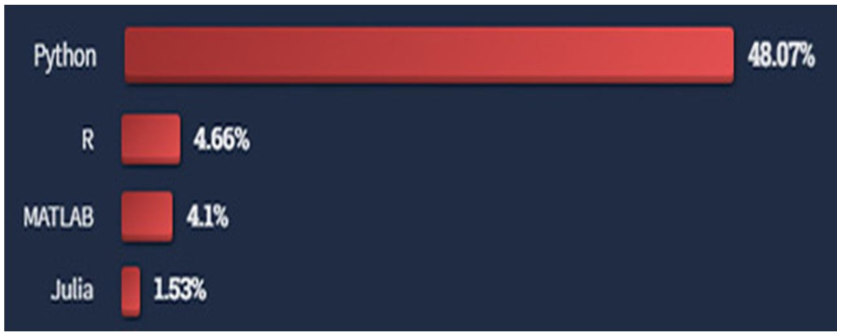

Fig. 2 The most popular programming language by 2022

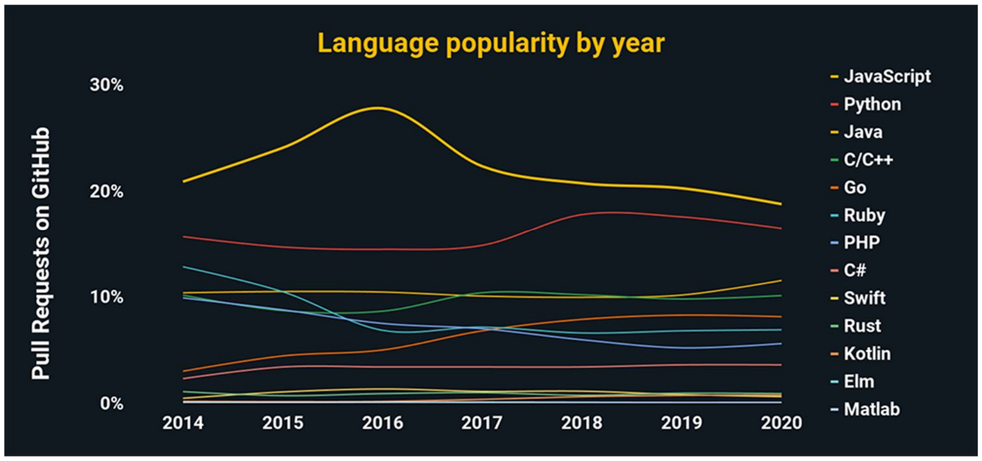

Fig. 3 different languages' popularity by year [10]

By 2021, about 10 million developers worldwide will be using Python [11]. Python has a large ecosystem of third-party libraries and tools, communities include various developer communities, and the python official website also provides detailed documentation and tutorials. But as Python continues to evolve, new releases may introduce some incompatible changes, and the quality and maintenance status of third-party libraries and tools may vary.
By 2020, MATLAB has more than 4 million users globally [12]. Matlab officially provides comprehensive and detailed documentation and tutorials, and Matlab is supported by MathWorks, which provides commercial support and training services. But compared to Python, Matlab is commercial software and requires the purchase of a license (cost about 6400 CNY per year) [13]. At the same time Matlab may be less efficient when dealing with large scale data. Also, Matlab is closed source software which limits its scalability.
As of 2020, Julia will be used by approximately 1.7million users worldwide [14]. Julia officially provides comprehensive and detailed documentation and tutorials and has a rich open-source library and package manager. Julia's ecosystem and user community is relatively small compared to Python and Matlab, and because Julia is relatively new, and some of the libraries and tools may not be as well documented or as well-resourced as other languages.
There are now about 2 million R users worldwide who utilize thousands of open sources packages within the R ecosystem in 2016 [15]. CRAN (Comprehensive R Archive Network) is the official package repository for the R language, with a rich set of third-party packages. R has a strong ecosystem in data science, and is a language focused on statistical analysis and graphical. But compared to Python and Matlab, R has a steeper learning curve. Since R is an interpreted language, it may have performance issues when working with large-scale data.
With a common interpreter, developers can take full advantage of Python's rich libraries and community, MATLAB's powerful numerical computation capabilities, R's focus on statistical analysis and graphing, and Julia's high-performance computing and ease of use. In summary, creating a common interpreter that accepts different syntaxes can provide developers with a unified development environment that integrates the strengths of various languages.

D. Debugging Levels
Python's debugging tools and levels are simple and easy to use, and provide detailed error information, which can help developers quickly locate and fix problems. python provides a variety of debugging tools, such as PDB, PyCharm. Python is difficult to debug complex problems, and developers may need a lot of time and workload to re-run the program to validate the fixes, and there is a risk of leakage of debugging information [5].
Matlab provides an interactive debugging environment with a rich set of built-in debugging tools and functions that help to quickly locate and solve problems. MATLAB's debugging tools rely heavily on the graphical interface, and Matlab's debugging capabilities are more limited compared to python. For beginners, Matlab debugging tools have a steeper learning curve [6].
The Julia debugger utilizes its dynamically typed and on-the-fly compilation features to provide efficient debugging functionality, and also provides an interactive debugging environment based on the REPL (Read-Eval-Print Loop). Since Julia is a relatively new programming language, its debugging tools and ecosystem may be relatively immature and unstable, while the learning curve is steep and community support may be limited [7].

R provides a rich set of debugging tools and functions, powerful data analysis and visualization capabilities, and a large user community. R may face some challenges when dealing with complex problems, especially for large-scale data processing, and may execute slower and have a steeper learning curve than python or Julia [8].
In summary, innovating a new interpreter that accepts different grammars can help to solve syntax transformation problems, improve development efficiency, and facilitate code sharing and collaboration. However, achieving a unified interpreter requires overcoming technical challenges and the complexity of unifying grammars, which requires a combination of factors to assess its feasibility and value.

E. Portability

Python is a cross-platform programming language that is widely used in a variety of domains, and also provides virtual environment management tools such as venv and conda. Python provides an integrated interface with the C/C++ language, which can be extended with modules to improve performance and functionality. Although Python itself has good portability, there may be some problems with compatibility of dependent libraries and modules [5].
Matlab also supports cross-platform, is widely used in a variety of fields, and provides a rich set of functions and toolboxes, but some domain-specific applications may require additional third-party libraries and tool support. Matlab is commercial software, and users are required to purchase a license to use it [6].
Julia is a cross-platform programming language with good interoperability with other programming languages such as Python, Matlab, and R. Julia has an active open-source community that provides a wealth of third-party libraries and tools covering a wide variety of domains and applications. Julia is relatively new, lacks some mature tools and libraries [7].
R is a cross-platform programming language with a wide range of applications in data analysis and statistical modelling. For beginners, R may have a steeper learning curve. R's packages and libraries may sometimes have dependencies and version updates may lead to incompatibility issues. This may require additional time and effort on the part of the user to manage and resolve dependencies [8].
Despite the portability of each of these languages, there are still some challenges and limitations. These include license restrictions, platform-specific dependencies, version compatibility, and performance issues. Based on these limitations, there are potential advantages of innovating an interpreter that accepts different syntaxes, which can provide greater portability and cross-platform compatibility, enabling developers to work more easily in different language environments. Such an interpreter may reduce the learning curve for developers between switching and adapting to different languages and toolsets and increase development efficiency, but it also faces a number of challenges and limitations.

F. Syntax Comparing

Table 1 shows the syntax analysis and comparison, we take four popular languages, Python, Matlab, Julia, and R, as examples.

	Python	MATLAB	Julia	R
Module	Modules or libraries #NumPy, Scipy, Pandas…	Built-in functions and toolboxes # User-defined functions and scripts	Ecosystem of packages # User Expandable Features	Ecosystem of packages #Developed and maintained by the R community.

Assignment operator	a = 3
	a, b =3, 5
		a = 3;
		[a, b] = deal(3, 5)’
			a = 3	a <- 3
				a = 3

Integer/Floating Point Selection	Type: int, float	Type: int, double	Type: int, Float64	Type：integer, numeric
	Automatically determined based on the value			
Dynamic type	Dynamic	Dynamic	Static	Dynamic

String	first_name = "John" last_name = "Doe" full_name = first_name + " " + last_name # or use join() full_name = ' '.join([first_name, last_name])	first_name = 'John'; last_name = 'Doe'; full_name = [first_name ' ' last_name];	first_name = "John" last_name = "Doe" full_name = "$first_name $last_name" full_name = first_name * " " * last_name	first_name <- 'John' last_name <- 'Doe' full_name <- paste(first_name, last_name, sep = " ") full_name <- paste0(first_name, last_name)
Built-in Boolean types	Boolean type: 'bool' Value: 'True' and 'False' '0' for 'False' and '1' for 'True'	Boolean type: 'logical' Value: 'true' and 'false' '0' for 'false' and '1' for 'true'	Boolean type: 'Bool' Value: 'ture' and 'false' No direct correlation with numbers	Boolean type: 'logical' Value: 'TRUE' and 'FALSE' '0' for 'False' and '1' for 'True'
Defining Multidimensional Arrays	Nested 'lists': matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] The NumPy library provides the ndarray type to create and manipulate multidimensional arrays.	Direct use of square brackets []: matrix = [1, 2, 3; 4, 5, 6; 7, 8, 9]	Direct use of square brackets []: matrix = [1 2 3; 4 5 6; 7 8 9]	Using the 'array' function: matrix <- array(c(1, 2, 3, 4, 5, 6, 7, 8, 9), dim = c(3, 3))
String (Slicing)	Substring = string[start:end]	Strsplit(string, delimiter)	Split(string,delimiter)	Strsplit(string,delimiter)
String (Indexing)	Char = string[index]	Char = string (index)	Char = string[index]	Char = string[index]
Array (Slicing)	Subarray = array[strat:end]	Subarray = array(start:end)	Subarray = array[start:end]	Subarray = array[start:end]
Array (Indexing)	Element = array[index]	Element = array(index)	Element = array[index]	Element = array[index]
Special Operator	**: Power operator	.*: Perform element-by-element multiplication of matrices or arrays	^: Power operator	^: Power operator
	//: Exact division operator	./: Performs element-by-element division of a matrix or array.	÷: Exact division operator	%%: Modal operator

	@: Matrix multiplication operator	.^: The dot-power operator	∘: Combining two functions into a new function	%*%: Performs multiplication of matrices
	-	@: Function handle operator	@: Calling Macros	%in%: Checks if an element is in a vector or list
Logical Operator	and	&&	&&	&&
	or	||	||	||
	not	~	!	!
{}	Creating dictionaries and sets	Creating cell arrays	Creating Dictionaries	Creating functions or control structures (e.g., code blocks)
[]	Creates a list or accesses elements of a list.	Creates matrices, vectors or accesses elements in a matrix/vector.	Create arrays, vectors or accessing elements of an array/vector	Used to access elements in a vector, list or matrix.
()	Function call, operator priority, creating tuple	Indicates a function call, control operator priority, or element in an indexed matrix/vector.	Indicates a function call, controls operator priority, or creates a list of function arguments.	Indicates a function call, controls operator priority, or creates a list of function arguments.
< >	Indicates a generic type or is used for comparison operators (e.g., less than, greater than).	Indicates inequality or is used to create anonymous functions.	Indicates a generic type or is used for comparison operators (e.g., less than, greater than).	Indicates unequal relationships or is used in control structures (e.g., for loops, while loops).

Syntax differences in matrix operations	import numpy as np
	
	# Creating matrices
	A = np.array([[1, 2], [3, 4]])
	B = np.array([[5, 6], [7, 8]])
	
	# Matrix multiplication
	C = np.dot(A, B) # or A @ B
	
	# Element-wise operations
	D = A + B
	E = A - B
	F = A * B # element-wise multiplication
		% Creating matrices
		A = [1, 2; 3, 4];
		B = [5, 6; 7, 8];
		
		% Matrix multiplication
		C = A * B;
		
		% Element-wise operations
		D = A + B;
		E = A - B;
		F = A .* B; % element-wise multiplication
			# Creating matrices
			A = [1 2; 3 4]
			B = [5 6; 7 8]
			
			# Matrix multiplication
			C = A * B
			
			# Element-wise operations
			D = A + B
			E = A - B
			F = A .* B # element-wise multiplication
				# Creating matrices
				A <- matrix(c(1, 2, 3, 4), nrow=2, ncol=2)
				B <- matrix(c(5, 6, 7, 8), nrow=2, ncol=2)
				
				# Matrix multiplication
				C <- A %*% B
				
				# Element-wise operations
				D <- A + B
				E <- A - B
				F <- A * B # element-wise multiplication

Process control	Conditional statements: if-elif-else Loops: for, while Exception handling: try-except	Conditional statements: if-else Loops: for, while	Conditional statements: If-else Loops: for, while Exception handling: try-catch	Conditional statements: If-else, switch Loops: for, while Exception handling: try-catch
Ternary operator	Ternary Operators: Python has ternary conditional operators (also known as ternary expressions) with the syntax expression1 if condition else expression2, which are used to select two different expressions to execute based on a condition. := (walrus operator): You can assign the value of an expression to a variable and use it in a conditional expression	-	? :: A ternary operator used to select two different expressions to execute based on a condition, with the syntax condition ? expression1 : expression2. ->: Define an anonymous function.	Special operators: %in%: membership operator, used to check if an element is in a vector or list. %>%: pipeline operator for conveniently combining multiple functions in a functional programming style.

Circulate	for:
	for item in iterable:
	  …
		for:
		for variable = range
		  …
		end
			for:
			for variable in collection
			  …
			end
				for:
				for(variable in sequence){
				  …
				}

	while:
	while condition:
	  …
		while:
		while condition
		  …
		end
			while:
			while condition
			  …
			end
				while:
				while(condition){
				  …
				}

	parfor:
	parfor variable = range
	  …
	end
		-	@simd:
			@simd for variable in collection
			  …
			end
				foreach:
				library(foreach)
				foreach(variable = sequence) %do%
				{
				  …
				}

Formatted output	name = “John”
	age = 25
	print(“My name is {} and I am {} years old.”.format(name, age))
		name = ‘John’;
		age = 25;
		fprintf(‘My name is %s and I am %d years old .\n, name, age);
		%: My name is John and I am 25 years old.
			name = “John”
			age = 25
			println(“My name is $name and I am $age years old.”)
				name <- “John”
				age <- 25
				sprint(“My name is %s and I am %d years old.”, name, age)

	print(f “My name is {name} and I am {age} years old.”)		# Formatting Strings with the @printf Macro @printf("My name is %s and I am %d years old.", name, age)	paste(“My name is”, name, “and I am”, age, “years old.”)

IV. CONCLUSIONS

By comparing the key points of four languages - Python, Matlab, Julia, and R - we came up with some important findings. First of all, Python has an advantage in terms of learning curve, its concise syntax and rich resources make it possible for beginners to get started quickly. Secondly, in terms of computational speed, Julia stands out with its high-performance computational capabilities, Matlab has powerful numerical calculations and a toolbox for specialized domains, while Python and R focus on flexibility and scalability. For community support, both Python and R have large and active communities that provide a wealth of resources and developer exchanges. Regarding debugging levels, both Python and Matlab provide powerful debugging tools and levels. As for portability, Python has extensive portability and third-party library support, while Julia focuses more on interoperability with other languages. Finally, in terms of syntax, each language has its own unique features and advantages for different application scenarios.

Based on the analysis of these findings, we believe there is value in creating a common interpreter that accepts different syntaxes. It would allow developers to choose different languages based on project needs, thus increasing development efficiency and flexibility. Despite some challenges, such as syntactic differences, they can be overcome through sound design and technical solutions. We look forward to more research and innovation to further advance the development of universal interpreters. This will promote communication and cooperation among different languages and bring more opportunities and development potential to the field of data analysis and scientific computing. Meanwhile, we also encourage developers and researchers to actively participate in the development and improvement of the Universal Interpreter and contribute to the progress of this field.

V. ACKNOWLEDGMENT

This paper was completed under the guidance of Professor Loskot, and we would like to thank our tutor Professor Loskot. Throughout our writing process, Professor Loskot gave us a lot of valuable advice. Professor Loskot has given us many valuable suggestions throughout the whole writing process. From the first idea to the revision of the thesis, Professor Loskot has been responsible for helping us to revise the thesis. We would like to take this opportunity to express our heartfelt thanks and respect for the completion of the paper.

REFERENCES

[1] Tsai, CW., Lai, CF., Chao, HC. et al, “Big data analytics: a survey,” Journal of Big Data., vol.1, Oct. 2015.
[2] (2022) Interpreter (computing). [Online]. Available: https://en.wikipedia.org/wiki/Interpreter_(computing)
[3] M. Šipek, B. Mihaljević and A. Radovan, “Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM,” 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, pp. 1671-1676, 2019.
[4] Dobiasd. (2014) Learning Curves (for different programming languages). [Online]. Available: https://github.com/Dobiasd/articles/blob/master/programming_language_learning_curves.md#learning-curves-for-different-programming-languages
[5] (2023) The MathWorks website. [Online]. https://ww2.mathworks.cn/products/matlab.html
[6] (2023) Python 3.12.2 documentation. [Online]. https://docs.python.org/3/
[7] (2023) Julia 1.10 Documentation. [Online]. https://docs.julialang.org/en/v1/
[8] (2023) The R Project for Statistical Computing. [Online]. https://www.r-project.org/
[9] (2022) 2022 Developer Survey. [Online]. Available: https://survey.stackoverflow.co/2022#most-popular-technologies-language

[10] N. Gallinelli. (2021) What Programming Language Should I Learn? [Online]. Available: https://flatironschool.com/blog/what-programming-language-should-i-learn/
[11] R. Daws. (2021) SlashData: JavaScript and Python boast largest developer communities. [Online]. Available: https://www.developer-tech.com/news/2021/apr/27/slashdata-javascript-python-boast-largest-developer-communities/
[12] (2022) MATLAB. [Online]. Available: https://en.wikipedia.org/wiki/MATLAB
[13] (2023) MATLAB pricing-licensing. [Online]. https://ww2.mathworks.cn/pricing-licensing.html
[14] (2023) Does anyone know an estimated number of Julia users? [Online]. https://discourse.julialang.org/t/does-anyone-know-an-estimated-number-of-julia-users/51711/4
[15] (2017) Is there a way to estimate the number of R users? [Online]. https://stackoverflow.com/questions/36529595/is-there-a-way-to-estimate-the-number-of-r-users