ARTURO MACHADO BURGOS

ASSESSMENT OF PROGRAMMING LANGUAGES FOR COMPUTATIONAL FLUID DYNAMICS

FEDERAL UNIVERSITY OF UBERLÂNDIA

SCHOOL OF MECHANICAL ENGINEERING

2021

ARTURO MACHADO BURGOS

Advisor

PROF. DR. JOÃO RODRIGO ANDRADE

ASSESSMENT OF PROGRAMMING LANGUAGES FOR COMPUTATIONAL FLUID DYNAMICS

Course Completion Project presented to the Undergraduate Course in Mechanical Engineering at the Federal University of Uberlândia, as a partial requirement to obtain the degree of **BACHELOR of SCIENCE**.

UBERLÂNDIA - MG

2021

ASSESSMENT OF PROGRAMMING LANGUAGES FOR COMPUTATIONAL FLUID DYNAMICS

Course Completion Project **APPROVED** by the Collegiate of Mechanical Engineering Undergraduate Course from the School of Mechanical Engineering at the Federal University of Uberlândia.

EXAMINING COMMITTEE

Prof. Dr. João Rodrigo Andrade
Federal University of Uberlândia

Prof. Dr. Ramon Silva Martins
Vila Velha University

Prof. Dr. Carlos Antônio Ribeiro Duarte
Federal University of Catalão

UBERLÂNDIA - MG

2021

ACKNOWLEDGMENTS

First of all, I would like to thank my family, Marco Antonio, my caring father, Taciana, my patient mother, and Lorenzo, my kind brother, for all their unconditional support and love during all those years.

A very special thanks to my colleagues from the 99th class of the Mechanical Engineering Course, to the co-workers of the Fluid Mechanics Laboratory, to the teammates of the Equipe Cerrado de Baja SAE and, especially, to my friends Guilherme Andrade, Iago Alves, João Paulo Gonzaga, João Pedro Vilela and Lucas Bento for all the times we had together and our friendship during the course; as well as to the professors Prof. Dr. Ana Marta de Souza, Prof. Dr. Daniel Dall'Onder and Prof. Dr. Aristeu da Silveira Neto for their advice and caring.

Last but not least, I would like to express my deepest gratitude to my advisor and mentor, Prof. Dr. João Rodrigo Andrade, for his guidance, friendship, patience, and knowledge which were undoubtedly essential for me to achieve my goal. Under his tutelage, he always supported and guided me on the sharpening of my skills as a future researcher. I also had the privilege to be one of his students in the Heat Transfer II course and, I can say that he was one of the best professors I had during my undergraduate course. Definitively, someone who I admire the most and a true friend.

“Machines take me by surprise with great frequency.”
(Alan Turing)

Burgos, A. M. **Assessment of Programming Languages for Computational Fluid Dynamics.** 2021. 62 p. Course Completion Project, Federal University of Uberlândia, Uberlândia, 2021.

ABSTRACT

For the vast majority resolution of both computational numerical dynamics and computational fluid dynamics, CND and CFD respectively problems, different programming languages are used. In this present work we sought to evaluate routines commonly used in CND and CFD codes to solve different problems. The level of complexity of the tests varies, ranging from easy such evaluation of loops or recursive problems to those at first somewhat complex, since they already deal with concepts of heat transport. Four programming languages were accounted for: Python, FORTRAN, Julia and Matlab. Furthermore, one of them contains a variation, which in this case Python with the Numba module (specifically Just In Time process - JIT). The analysis of the languages are both quantitative and qualitative. That is, despite worrying about analyzing the computational time spent on each of the tests, we also try to show our impressions about the ease of use or the documentation available in forums. As for the results, it is possible to infer that among the programming languages the one with the best performance is FORTRAN. However Python demonstrates to have a great potential, since it is a high-order language and very embraced by the community. In addition, it was found that with routines created by other developers, processes are greatly optimized, making execution times even shorter when compared to the other languages.

Keywords: programming languages; computational; computational fluid dynamics; vectored process; computational numerical dynamics

Burgos, A. M. **Assessment of Programming Languages for Computational Fluid Dynamics.** 2021. 62 f. Trabalho de Conclusão de Curso, Universidade Federal de Uberlândia, Uberlândia, 2021.

RESUMO

Para a resolução da grande maioria dos problemas de dinâmica numérico-computacional quanto a dinâmica de fluidos computacional, CND e CFD, respectivamente, diferentes linguagens de programação são usadas. No presente trabalho buscamos avaliar rotinas comumente utilizadas em códigos CND e CFD para resolver diferentes problemas. O nível de complexidade dos testes varia, indo desde fáceis como avaliação de loops ou problemas recursivos até aqueles inicialmente um tanto complexos, uma vez que já lidam com conceitos de transporte de calor. Quatro linguagens de programação foram contabilizadas: Python, Fortran, Julia e MATLAB. Além disso, um deles contém uma variação, neste caso Python com o módulo Numba (especificamente o processo Just In Time - JIT). As análises das linguagens são quantitativas e qualitativas, ou seja, apesar de nos preocuparmos em analisar o tempo computacional despendido em cada um dos testes, procuramos também mostrar nossas impressões sobre a facilidade de uso ou sobre a documentação disponível nos fóruns. Quanto aos resultados, é possível inferir que dentre as linguagens de programação a que apresenta melhor desempenho é o FORTRAN. Porém Python demonstra ter um grande potencial, visto que é uma linguagem de alto nível e muito aceita pela comunidade. Além disso, verificou-se que com rotinas criadas por outros desenvolvedores, os processos são bastante otimizados, tornando os tempos de execução ainda mais curtos quando comparados aos demais.

Keywords: linguagens de programação; computacional; fluidodinâmica computacional; processos vetorizados; dinâmica numérico-computacional

List of Figures

1 Golden spiral representing Fibonacci's 10
2 Surface Scheme with each boundary temperature condition. 13
3 Surface Scheme with the mesh condition 14
4 Stencil Scheme, fundamental unity of the mesh, with one central node and four peripheral nodes 15
5 Surface Scheme with the mesh condition where same colors represent similar shape construction 16
6 Execution time for Test 1 by applying loop operations, $n = 3000$ is the matrix coordinate elements number. 21
7 Execution time for Test 1 by applying loop operations, $n = 5000$ is the matrix coordinate elements number. 22
8 Execution time for Test 1 by applying vectored operations, $n = 3000$ is the matrix coordinate elements number. 23
9 Comparative between elapsed time of two different python implementation . . 24
10 Execution time for Test 1 by applying vectored operations, $n = 5000$ is the matrix coordinate elements number. 24
11 Execution time for Test 2 by applying loop operations, $n = 250$ is the square matrix order and it was assigned for all languages. 25
12 Execution time for Test 2 by applying loop operations, $n = 350$ is the square matrix order and it was assigned for Matlab, Python - JIT and FORTRAN. . . . 26
13 Execution time for Test 2 by applying vectored operations, $n = 1500$ is the square matrix order and it was assigned for all languages, with the exception of Python - JIT for which matrix multiplication modules are not supported. 27
14 Execution time for Test 2 by applying vectored operations, $n = 5000$ is the square matrix order and it was assigned for Matlab and FORTRAN. 28
15 Execution time for Test 3 by applying iterative operations, $n = 30$ is the index series number for all languages. 29
16 Execution time for Test 3 by applying iterative operations, $n = 45$ is the index series number for all languages. 30
17 Comparative execution time for Test 3 by applying iterative variable and array operations, $n = 45$ is the index series number for Python, Julia and FORTRAN. 30
18 Execution time for Test 3 by applying recursive operations, $n = 30$ is the index series number for all languages. 31
19 Execution time for Test 3 by applying recursive operations, $n = 38$ is the index series number for all languages. 32

20 Execution time for Test 3 by applying recursive operations, $n = 38$ is the index series number for Python - JIT, FORTRAN and Julia. 32
21 Execution time for Test 4, $n = 10000$ is the slots in each array for Python - JIT, FORTRAN, Julia and Matlab. 33
22 Execution time for Test 4, $n = 1000000$ is the slots in each array for Python - JIT, FORTRAN, Julia and Matlab. 34
23 Execution time for Test 5 by applying loop operations, $n = 8x8$ is the mesh for all languages. 35
24 Execution time for Test 5 by applying loop operations, $n = 12x12$ is the mesh for all languages. 36
25 Execution time for Test 5 by applying loop operations, $n = 14x14$ is the mesh for all languages. 36
26 Execution time for Test 5 by applying loop operations, $n = 25 \times 25$ is the mesh for Python - JIT, Matlab and FORTRAN. 37
27 Matlab contour plot - Surface temperature distribution for steady regime, mesh - 8x8 38
28 Matlab contour plot - Surface temperature distribution for steady regime, mesh - 12x12 38
29 Matlab contour plot - Surface temperature distribution for steady regime, mesh - 14x14 38
30 Matlab contour plot - Surface temperature distribution for steady regime, mesh - 25x25 38
31 Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 8x8 39
32 Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 12x12 39
33 Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 14x14 39
34 Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 25x25 39
35 Example of execution time for Python using linalg/LAPACK method, various n values. 40
36 Matrix of standardized elapsed times, where as the colors become darker, it indicates that there is the shortest run-time 41

List of Tables

1 First Test - Comparison Loop Test for n = 3000 48
2 First Test - Comparison Loop Test for n = 5000 48
3 First Test - Comparison Vectored Test for n = 3000 48
4 First Test - Comparison Vectored Test for n = 5000 48
5 Second Test - Comparison Loop Test for n = 250 49
6 Second Test - Comparison Loop Test for n = 350, Python n = 250 49
7 Second Test - Comparison Vectored Test for n = 1500 49
8 Second Test - Comparison Vectored Test for n = 5000, Python and Julia n = 1500 49
9 Third Test - Comparison Iterative Test for n = 30 49
10 Third Test - Comparison Iterative Test for n = 45 50
11 Third Test - Comparison Between Variable and Iterative Array for n = 45 . . . 50
12 Third Test - Comparison Recursive Test for n = 30 50
13 Third Test - Comparison Recursive Test for n = 38 50
14 Fourth Test - Comparison for n = 10000 50
15 Fourth Test - Comparison for n = 1000000 51
16 Fifth Test - Comparison Loop Test for n = 8x8 51
17 Fifth Test - Comparison Loop Test for n = 12x12 51
18 Fifth Test - Comparison Loop Test for n = 14x14 51
19 Fifth Test - Comparison of Matlab, Python - JIT and FORTRAN for n = 25x25 51
20 Example of execution time for Python using linalg/LAPACK method, various n values 52

Contents

1 Introduction **2**

2 Methodology **6**

2.1 First Test - Copying Arrays 6

2.2 Second Test - Matrix Multiplication 8

2.3 Third Test - Fibonacci's Sequence 9

2.4 Fourth Test - Writing information in a File 12

2.5 Fifth Test - Solving Diffusion Equation 13

3 Results **20**

3.1 First Test 21

3.1.1 Loop Operations 21

3.1.2 Vectored Operations 23

3.2 Second Test 25

3.2.1 Loop Operations 25

3.2.2 Vectored Operations 26

3.3 Third Test 28

3.3.1 Iterative Operations 29

3.3.2 Recursive Operations 31

3.4 Fourth Test 33

3.5 Fifth Test 34

3.6 Overall results 41

4 Conclusion and Further Work **43**

Annex **48**

1 Introduction

Programming language is a formal language that includes a series of instructions for generating various types of outputs. They are used to write programming codes to perform a message that is sent to the machine, which is the algorithm. In other words, it consists of commands for the machine to obey. Like all other spoken and written languages, the programming language has its own grammar and semantics. Nowadays, there are a lot of them, but it is up to the programmer to pick the language he is going to use to write his codes.

There are many uses for programming in the engineering field. Several challenges most engineers face through their work can be overcome by programming and applying a logical algorithm. A basic example will be the resolution of certain differential equations: it is not difficult to find mathematical models that are nearly impossible to solve by empirical, but only through a numerical computational method. Knowing that, it is easy to realize how necessary it is for an engineer to be able to program, in order to solve a particular problem, when this approach is required.

However, as mentioned at the outset, there are a lot of different programming languages on the market. They vary primarily in syntax and grammar from each other. Taking syntax into account, there are basically two types of languages: high-level and low-level ones. High-level language is a programming language that is more distant from the specifics of the machine. To put in another way, it is a more understandable language, in which features and definitions are easy to use. On the other hand, while the low-level programming language appears to be more complex for a normal person, the fact that it is similar to the assembly language (machine code) is generally the explanation why it also runs faster.

Normally, students get in touch with at least two separate programming languages in most engineering classes. However, one of them is most likely to be Matlab. It is used to teach the first principles of an algorithm and is probably the most widespread language in the entire engineering academy due to its inherent connection with the language of C and C++ and its extensive number of libraries. Nevertheless, in computational dynamics such as CFD programming, owing to its fast results in science computing, most engineers tend to use the FORTRAN language. Therefore, there are certain things to be taken into account when picking the best programming language in order to solve a problem, as imagined.

However, establishing efficient evaluations and comparisons of different programming languages is not a trivial task. Several authors have contributed to the development of useful works, such as Domke [2020], Raschka [2014], Kouatchou [2019], Anik and Baykoç [2011]. Kouatchou [2019] postulates that focusing on the analysis of how fast a given language is, is not

a concrete analyzes method regarding the capacity of the language or the machine that executes the code. Some works like Pereira et al. [2017] even presents a co-relation between power consumption, memory usage and the elapsed time of each analyzed languages. Having this in mind, both engineers and programmers could decide which is the best programming language in order to solve several problems concerning energy efficiency.

It is worth mentioning that comparing different languages is not as simple as to analyze its performance. However, the real comparison remains on finding which is the most efficient language in order to solve a specific problem - additionally with its supporting modules, and the easiness to understand. For instance, Kouatchou [2019] analyzed several languages in relation to their efficiency in vectorization, regular loops and intrinsic routines.

In the present study, we seek to compare computational languages usually applied in numerical calculations to solve problems in computational fluid mechanics, which are:

1. FORTRAN 90
2. Julia v1.0.5
3. Python 3.6.8
4. Python + NUMBA v0.51.1
5. Matlab v2018b

The criterion used to select the above programming languages was: in relation to Matlab, we used the v2018b because Federal University of Uberlândia holds its license. The others have a free license, then we had preference for each newest version. It is also important to mention that no compilation flags were taken into consideration in the studied cases. A brief summary of each used programming language is available below.

According to Fehr and Kindermann [2018], FORTRAN, from the original, FORMULA TRANSLATION is a large widely used language, mainly for numerical computing or for scientific computing. Originally developed at the IBM laboratory during the 50's, it has mastered programming since its creation, being used even today in computational fluid dynamics (CFD), numerical approximations, computational physics and computational chemistry.

Julia is a high-performance language as it is said in the JuliaLang Organization [2021] documentation. Its common nickname of "Language of the Future" is due to its ability to be as efficient as C and because of it being a high-level language like Python. Therefore, Julia unites the best of both worlds. Although used for different applications, Julia's common use is directed to numerical or scientific analysis, like FORTRAN. The license is from MIT and the language was developed between 2009 and 2012.

Python, unlike previous languages, is not compiled but interpreted. Also a high level language, it was originally developed by Guido van Rossum in 1991. Nowadays, community development by Python Software is a non-profit organization. Being considered as the third most loved programming language in the world, we see its great growth. As a curiosity, its name comes from the famous British comedy series Monty-Python. There is no preference for its use, as Guttag [2016] proposes is a "general-purpose programming language that can be used effectively to build almost any kind of program that does not need direct access to the computer's hardware". Although it is currently being widely used for data-science. Incorporating the NUMBA module with JIT ("Just In Time" operations), Python's efficiency increases considerably (as well as the time spent on processes and sub-processes). According to Numba Pydata Organization [2021], Numba is an open source JIT compiler that converts a subset of Python into fast machine code using LLVM, through the low-level virtual machine package, in this particular case C++.

Matlab (MATrix LABoratory) is a high-performance software focused on numerical analysis. According to Backes [2013], Matlab was originally developed as a module of the C language. It also integrates matrix calculation, signal processing and graphic construction. It is an interactive system whose basic information element is a matrix that does not require dimensioning, that is, it is fully vectored. That condition allows the resolution of many numerical problems in just a fraction of the time it would take to write a similar program in non-vectored languages. The solution/output of the programs is usually expressed in the way it was proposed, which is mathematically. There are some vast documentations, including books such as Danaila [2007] that discuss numerical implementations of different numerical problems. It is worth remembering the existence of GNU Octave, a software that looks like Matlab in essence but is open-source. At least, at the time this work has been written Matlab is still a payed software.

For all the above listed languages there are some software and CFD solvers. In relation to FORTRAN it is possible to cite the Fluid Mechanics Laboratory of Federal University of UberLândia (MFLab) In-House code called *MFSim*. Multiple articles from MFLab have been published using the numerical simulations of this software. Both Python and Matlab have external optimized modules that perform CFD calculations. As an example, it is possible to cite Mohanan et al. [2019] and Precise Simulation [2021] as complete libraries for Python and Matlab respectively. Last but not least, Pawar and San [2019] postulate CFD foundations using Julia Language in order to create solvers for specific problems.

The present paper, initially, aims to establish comparisons of different programming languages by applying them to specific mathematical and numerical procedures. In addition to the languages themselves, several parameters are also analyzed, e.g, the effects of vectoring variables instead of using loops. The tests performed here are implicitly linked to any algorithm applied in Computational Fluid Dynamics problem. Therefore, those routines were selected to

be evaluated in the above mentioned programming languages. All tests were done in isolation so that there was no interference from third parties in the speed. The computer parameters are listed below:

- Operational System: Ubuntu 18.04.4 LTS
- Processor: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz x4
- Installed RAM: DDR3 (800 to 2400 MHz rate) 15,3 GB
- Architecture: x86_64

2 Methodology

The parallels indicated in the present work discuss routines commonly used in CFD algorithms, i.e. solving linear equation scheme and shifting locations of vector components. There are five tests: four of which can be considered as simple tests behind which there are no mathematical formalization behind them. However, the last test is required to show the steps of the resolution. The tests are Copying Arrays, Matrix Multiplication, Fibonacci's Sequence, Writing information in a file and Solving a System of Linear Equations (2D Diffusion Equation). These are critical experiments since it is possible to set up a mesh and control vast volumes of data in major CFD and heat transfer topics.

The development of each section will follow this lead:

- first we describe each problem and how to solve it,
- then the pseudo-code is shown in order to provide an indication of the resolution.

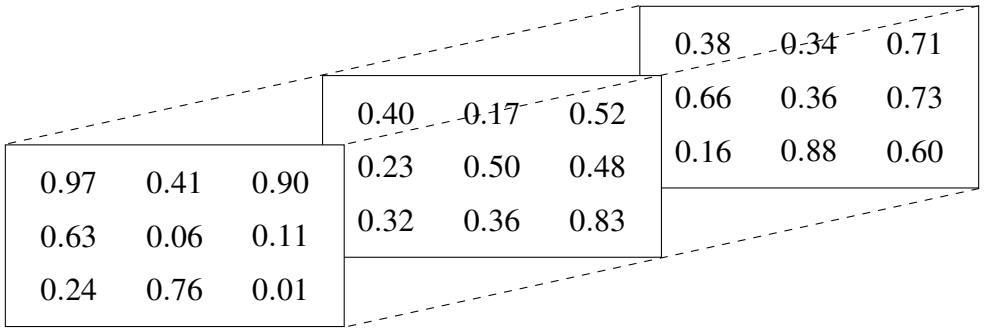
Although the language's syntax are different from each other, they follow a pattern. With the support of Visual Studio, a free-source code editor available for Linux, Windows and macOS, the codes were written. All the codes developed in this present work can be found in the author's personal GitHub repository, available at: https://github.com/arturofburgos/

2.1 First Test - Copying Arrays

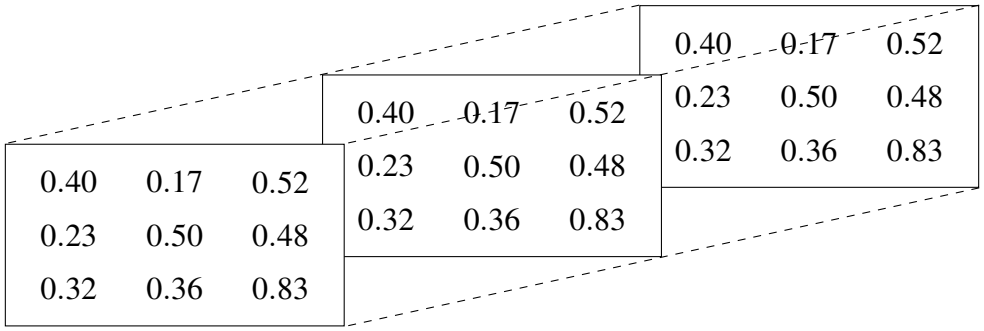
At the outset, as the first test, the problem relates to thinking a random A tridimensional matrix—shape $(n,n,3)$, where n is the dimension of i and j. We are going to change this same matrix in this test according to the following equation:

$$A[i,j,1] = A[i,j,2]$$
$$A[i,j,3] = A[i,j,1]$$

This test is important because, in addition to dealing with iterative loops, it also deals with three-dimensional matrices, assignment and values exchange, all of which are routines that are common in computational fluid dynamics problems. As an example, suppose we have a random given matrix B $(n,n,3)$ and, to simplify, let us make $n = 3$. So, B is given by:



Where, each cell contending a bi-dimensional matrix represents the third dimension, which is what makes it 3D in the coordinate Z. So, applying B to the above equations we will find as a result:



Now, regarding the algorithm, there are two ways of doing so: by using a common loop or with the usage of vectorization. A shorter elapsed time is expected when applying vectorization due to the fact that it is an intrinsic feature of each language. Putting it in another way, instead of dealing with one element at a time (which happens in the common loop), vectorizing a task allows to process, let's say, 4 or more elements at once. For that, the numerical operations have been made by using iterative loop and vectored procedure. It is expected that the latter one will have shorter runtimes.

The pseudo-code for the vectored procedure is shown below:

Algorithm 1 First Test - Use of Vectorization

Initialize a random 3D matrix A NxNx3

Vectorization function
$A[:,:,1] \leftarrow A[:,:,2]$
$A[:,:,3] \leftarrow A[:,:,1]$
return A

set initial time
Vectorization
set final time

Now the pseudo-code for the loop test is given by:

Algorithm 2 First Test - Use of Loop

Initialize a random 3D matrix A NxNx3

Loop function
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**
      $A[i, j, 1] \leftarrow A[i, j, 2]$
      $A[i, j, 3] \leftarrow A[i, j, 1]$
    end for
  end for
return A

set initial time
Loop
set final time

2.2 Second Test - Matrix Multiplication

The goal of this next test is to solve a simple matrix multiplication. Assuming an A and B arbitrary bi-dimensional matrix - square shape (n, n), where n is also the size i and j dimension of each. Mathematically speaking the multiplication is given by the formula:

$$R = A \cdot B \tag{1}$$

Where R is the result of this operation. For instance, we have two random 3x3 matrices:

$$\begin{bmatrix} 14 & 10 & 11 \\ 18 & 14 & 18 \\ 12 & 11 & 17 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 3 \\ 4 & 2 & 4 \\ 1 & 4 & 3 \end{bmatrix} \times \begin{bmatrix} 2 & 1 & 2 \\ 1 & 1 & 3 \\ 2 & 2 & 1 \end{bmatrix}$$

Just like in the previous test, it is important to mention that some programming languages have this process as an intrinsic routine. Therefore, it is anticipated, that the vectored technique would also greatly reduce the run-up time. The pseudo-code for the vectored method is very simple and is given by:

Algorithm 3 Second Test - Use of Vectorization

Initialize a random matrix A and B

Vectorization function
    $R \leftarrow A * B$
return $R[i, j]$

set initial time
Vectorization
set final time

On the other hand, the loop test is considerably more complicated, since to solve this task it is necessary to define the variable *soma*. It will be initialized to zero and the sum of each step will be attributed in a third iteration, which will represent the index that both the column of matrix A and the line of matrix B must have in common so that the multiplication can be done. The pseudo-code is generated by:

Algorithm 4 Second Test - Use of Loop

Initialize a random matrix A and B

Loop function
    for $i = 1$ to n **do**
        for $j = 1$ to n **do**
            soma = 0
            for $k = 1$ to n **do**
                $soma \leftarrow soma + A[i,k] * B[k, j]$
                $R[i, j] \leftarrow soma$
            end for
        end for
    end for
return $R[i, j]$

set initial time
Loop
set final time

2.3 Third Test - Fibonacci's Sequence

The next test of this work was to measure the outcome of Fibonacci's series, given some n indices. Fibonacci's first appearance in western cultures was in Leonardo Fibonacci's book Liber Abaci (1202), in which he considered the rise of an idealized (not biologically realistic) population of rabbits. Mathematically speaking, the infinite sequence of numbers is governed by

the following formula:

$$F_n = F_{n-1} + F_{n-2} \tag{2}$$

$$\text{Where } n \geq 1 \in I.$$

That generates the following sequence:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 ...

The sequence is very important and has applications in financial markets analysis, computer science and game theory. The most curious aspect of it is the relation between the golden ratio and, because of that, the golden spiral - normally the first thing we associate when dealing with this problem.

Figure 1 shows the golden spiral, which is created by drawing circular arcs connecting opposite corners of squares in the Fibonacci tiling:

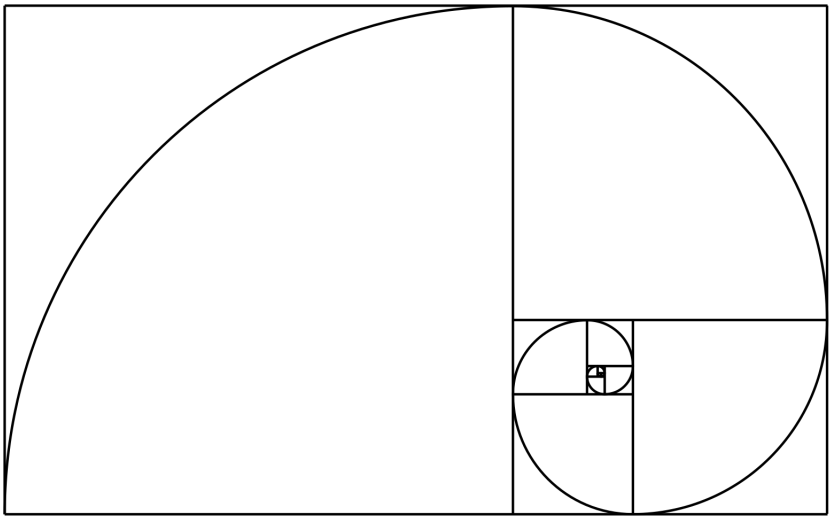


Figure 1: Golden spiral representing Fibonacci's

The key aim here is to measure the execution time of the Fibonacci program according to its *n* indices. However, for this third test, we used iterative loops and recursive functions to address this issue. A recursive function, by definition, is a function that calls itself during its execution so that execution time is supposed to last longer than normal loops. A significant comment about loops is that there are two separate approaches: by assigning three variables and by using an array.

The first written code was connected to the easiest test: the vector loop in this case. The

pseudo-code below gives an idea of how to proceed with this problem:

Algorithm 5 Third Test - Use of Loop - variables

Initialize the variables

Var1 = 0
Var2 = 1
Fbn = 0

Loop function(k,F1,F2,S)
    for $i = 1$ to k **do**
        $F1 \leftarrow S$
        $S \leftarrow F1 + F2$
        $F2 \leftarrow F1$
    end for
return S

set initial time
Loop function(n,Var1,Var2,Fbn)
set final time

As for the array, we have seen a very similar mechanism here that also involves a loop. The modification here is that we used an intrinsic function called "sum", which calculates the summation of the array in each iteration. Due to the use of recursion, a longer time spent running this code is expected. The algorithm is seen below:

Algorithm 6 Third Test - Use of Loop - array

Initialize the variable and array

A = array(0,1)
Fbn = 0
Loop function(k,F,S)
    for $i = 1$ to k **do**
        $F(1) \leftarrow S$
        $S \leftarrow sum(F)$
        $F(2) \leftarrow F(1)$
    end for
return S

set initial time
Loop function(n,A,Fbn)
set final time

Finally, we dealt with the recursive mechanism of the problem. In general, although considered more efficient in terms of witting, due to the use of a minimal amount of code, it is expected to take more time to be executed. The algorithm is seen below:

Algorithm 7 Third Test - Use of Recursion

```
Recursive function(n)
    if i ≤ 2 then
        return n
    else
        return Recursive function(n − 1)
    end if

set initial time
Fbn = Recursive function(n)
set final time
```

2.4 Fourth Test - Writing information in a File

This next test, in fact, is very simple. Basically, we created an X array with size n. Each position of the array contained the index number assigned to it. As an example, if we assign the value 5 to the parameter n, we would have this array:

$$X = [1\ 2\ 3\ 4\ 5]$$

After that, we applied the same array into some intrinsic function. In that case we used the cosine intrinsic function for each programming language. Using the array above, the result would be:

$$Y = [0.5403\ -0.4161\ -0.9899\ -0.6536\ 0.2836]$$

Then, we created a *.txt* file and wrote both arrays on it. As in the previous experiments, we also measured the elapsed time of the execution. The algorithm is shown below:

Algorithm 8 Fourth Test

```
Initialize the X and Y array

X = [1:1:n]
Y = cos.(X)

.txt function(Var1,Var2)
    file = OPEN(path)
    file.write(Var1)
    file.write(Var2)
    file.close()

set initial time
.txt function(X,Y)
set final time
```

2.5 Fifth Test - Solving Diffusion Equation

As the last test, the solution of a linear system based on the following problem is proposed. The solution is divided into mathematical and numerical model. Basically, it consists of a square and solid flat surface (one meter each size), with specified boundary conditions, and here our aim is to set the internal temperature of the surface by means of a numerical method. Also, it is important to mention that here, our goal is not to define all the mathematical formalization of the problem, but just to give a brief context and solve it. It is easy to see that the surface temperature is determined by a combination of each near boundary temperature shown in Fig 2, where a better visualization of the case domain is available.

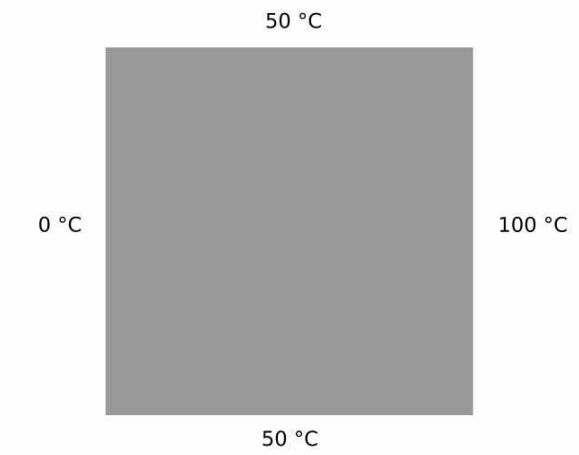


Figure 2: Surface Scheme with each boundary temperature condition.

In order to solve the mathematical model first, we determined the temperature equation that governed this model. In this case, according to White [1991], it is derived from the energy equation and intimately associated to the first law of Thermodynamics:

$$\rho c_P \frac{DT}{Dt} = k\nabla^2 T + \phi(\vec{V}) + S_i \tag{3}$$

And the total derivative term, $\frac{DT}{Dt}$, can be expanded into:

$$\rho c_P \left[\frac{\partial T}{\partial t} + \vec{V} \cdot (\vec{\nabla} T)\right] = k\nabla^2 T + \phi(\vec{V}) + S_i \tag{4}$$

However, aiming the simplification of the model we have proposed some hypothesis:

- As mentioned earlier it consists of a solid surface. Then, as a consequence there is no velocity field in this model.

- This is a steady regime, so there is no dependence on time. This means that the partial derivative with respect to time is equal to zero.
- There are only constant properties.
- The equation is solved in two dimensions.
- Finally yet importantly, there is no source term, such as internal heating.

With all the hypotheses put foward, Eq. 4 can be re-written as the diffusion equation:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{5}$$

Before the numerical approach, we have to deal with the discretization of the domain and the creation of the computational grid. Figure 3 shows the mesh where we divided the square flat surface into several small volumes, defined by parallel (in both vertical and horizontal) lines, configuring a structured mesh. The stencil, the fundamental part of the mesh, is essentially composed of five volumes which can be seen in the Fig. 4. It is worth mentioning that both Δx and Δy are equal in size and, indices i and j refers to x and y coordinates respectively.

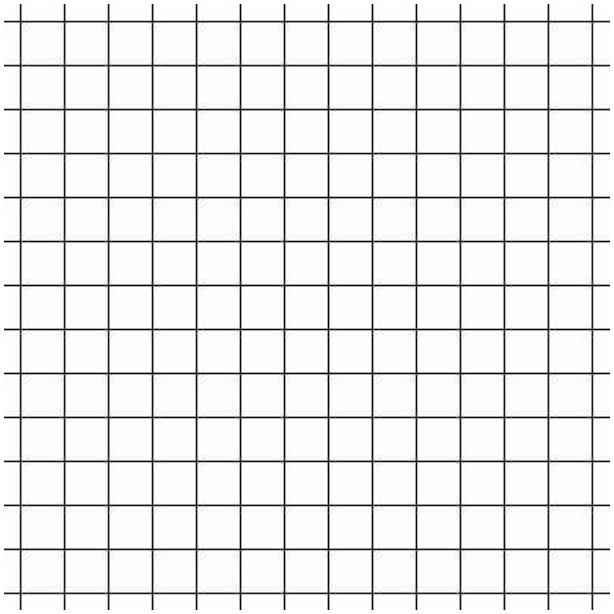


Figure 3: Surface Scheme with the mesh condition

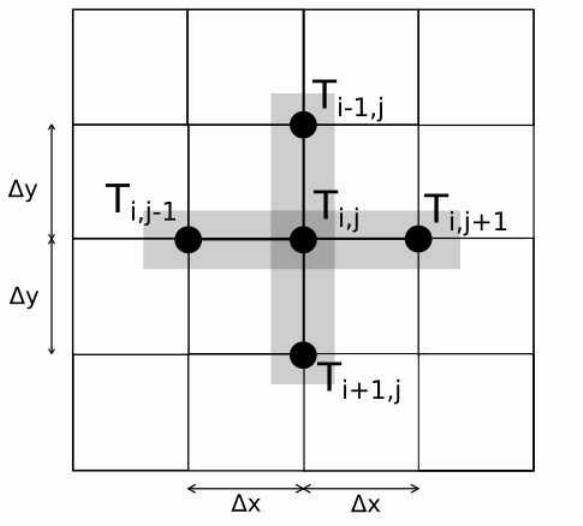


Figure 4: Stencil Scheme, fundamental unity of the mesh, with one central node and four peripheral nodes

Then, we applied the principle of finite differences, more precisely the central difference. According to Ferziger [2020], in partial differential equations concerned with second derivative terms, it is more fitting to use a central difference (a mixture of forward and backward differences) where the truncation error $O(x^2)$ and $O(y^2)$, coming from the Taylor series expansion, are smaller than the others. Thus, the terms on the left side of the eq. 5 can be discretized as:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{\Delta x^2}$$

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{\Delta y^2}$$

As mentioned earlier Δx and Δy are equal in size, so when adding both and making that sum as equal to zero, we encounter the discrete form of the previous eq. 5, which is given by:

$$T_{i-1,j} - 2T_{i,j} + T_{i+1,j} + T_{i,j-1} - 2T_{i,j} + T_{i,j+1} = 0 \tag{6}$$

And re-arranging eq. 6:

$$-4T_{i,j} + T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1} = 0 \tag{7}$$

$$T_{i,j} = \frac{T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1}}{4} \tag{8}$$

Basically, this shows that the temperature of each node or volume $T_{i,j}$ will be represented by the arithmetic mean of the peripheral terms of the stencil shown in the Fig. 4.

Regarding the boundary conditions, all of them are imposed temperatures, thus representing those known as Dirichlet. As shown in Fig. 5, each number represents a region of the square surface. Only the region 9 corresponding to the internal nodes is not directly affected by the boundary temperatures.

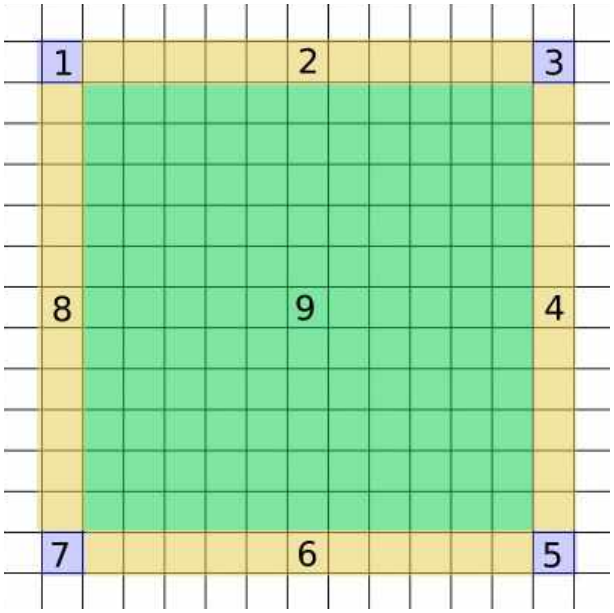


Figure 5: Surface Scheme with the mesh condition where same colors represent similar shape construction

Regions 2, 4, 6 and 8 have very similar shaping in relation to the associated terms. The same happens to the odd numbers excluding 9. In order to make this concept more understandable, taking the first node, represented by the northwest corner, its temperature is computed as:

$$T_{i,j} = \frac{0 + 50 + T_{i,j+1} + T_{i+1,j}}{4} \tag{9}$$

Using the same reasoning, the temperature of an internal point, corresponding to the green region can be represented by eq. 8. At the same time, a node located in an orange region, such as one in the upper edge (2) can be define by:

$$T_{i,j} = \frac{50 + T_{i,j+1} + T_{i+1,j} + T_{i+1,j-1}}{4} \tag{10}$$

Knowing that, it is easy to find the equation for temperature at each node, based on their location through the numbers or colors in Fig 5.

Therefore, in this case, we have a linear system of equations, which is given by:

$$b = A \cdot T \tag{11}$$

Where,

- b, a shape $(n, 1)$ matrix, represents the combination of the boundary and internal conditions
- A, a shape (n, n) matrix, represents the coefficients
- T formalizes each node of the grid

In order to help with the understanding, one can find bellow the visualization of those matrices when considering a two-dimensional space (n^2).

$$\begin{bmatrix} -50 \\ \vdots \\ -150 \\ 0 \\ \vdots \\ 100 \\ -50 \\ \vdots \\ -150 \end{bmatrix} = \begin{bmatrix} -4 & 1 & 0 & \cdots & 1 & 0 & \cdots & & 0 \\ 1 & -4 & 1 & & & & & \ddots & \\ 0 & 1 & -4 & 0 & & & \ddots & & \vdots \\ 1 & & 0 & -4 & & \ddots & & & \\ 0 & & & & \ddots & & & & 0 \\ & & & \ddots & & -4 & 0 & & 1 \\ \vdots & & \ddots & & & 0 & -4 & 1 & 0 \\ & \ddots & & & & & 1 & -4 & 1 \\ 0 & & \cdots & 0 & 1 & \cdots & 0 & 1 & -4 \end{bmatrix} \cdot \begin{bmatrix} T_{1,1} \\ \vdots \\ T_{1,n} \\ T_{2,1} \\ \vdots \\ T_{n-1,n} \\ T_{n,1} \\ \vdots \\ T_{n,n} \end{bmatrix}$$

Aiming at the resolution of this linear system of equations, several methods may be applied. However, we chose the Gauss-Seidel approach. Burden and Faires [2015] postulate that this solving method contains a stop criteria, a given first guess and defines $\{x^{(k)}\}_{k \geq 0}$ according to the given equation:

$$T_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} A_{ij} T_j^{k+1} - \sum_{j=i+1}^{n} A_{ij} T_j^k}{A_{ii}} \tag{12}$$

Since the previous concepts were stated, the next step is to write the algorithms in the chosen programming languages. The first one related to the matrices creation followed by the second one, which is the solution of the linear system itself. Both are shown below:

Algorithm 9 Fifth Test - Matrix A and b definition

```
Initialize n
Initialize k = sqrt(n)

Create Matrix A:
    for i = 1 to n do
      for j = 1 to n do
            if i = j then
                  A(i, j) ← −4
            else if (i = j − k) or (i = j + k) then
                  A(i, j) ← 1
            else if (mod(i, k) ≠ 0 and (i = j − 1)) or (mod(i, k) ≠ 1 and (i = j + 1)) then
                  A(i, j) ← 1
            else
                  A(i, j) ← 0
            end if
      end for
    end for

Create Matrix b:
    for i = 1 to k do
      if i ≤ k then
            b(i) ← −50
      else
            b(i) ← −150
      end if
    end for
    for i = k + 1 to n − k do
      if mod(i, k) ≠ 0 then
            b(i) ← 0
      else
            b(i) ← −100
      end if
    end for
    for i = n − k + 1 to n do
      if i ≤ n then
            b(i) ← −50
      else
            b(i) ← −50
      end if
    end for
```

Algorithm 10 Fifth Test - Linear System

Initialize n
Initialize k = $sqrt$(n)
Initialize Matrix A
Initialize Matrix b

linearsystem function(coeff,resul,size)
Set tolerance and first guess

erro = norm$(x^{k+1} - x^k)$
while erro $\leq tol$ **do**
for $i = 1$ to n **do**
$x^{k+1}(i) = b(i)$
for $j = 1$ to n **do**
if $j \neq i$ **then**
$x^{k+1}(i) \leftarrow x^{k+1}(i) - a(i,j) \cdot x^k(j)$
end if
end for
$x^{k+1}(i) \leftarrow x^{k+1}/a(i,i)$
end for

erro = norm$(x^{k+1} - x^k)$
$(x^k \leftarrow x^{k+1})$
return x^{k+1}

set initial time
T = linearsystem function(A,b,n)
set final time

3 Results

In this section, we aim to identify the programming language that perform best. For this purpose, we consider the elapsed time as an analysis parameter. In addition, some comments and impressions about the user experience will be given, since some specific issues of each language were encountered.

Although there are different and, even more, intrinsic optimized procedures to solve the proposed problems, our goal here was to simulate an environment in which no programming language had an advantage over another. To put in other words, external packages or extremely optimized third party routines were completely neglected during algorithm construction. As a simple example, we could cite the *linalg* package from Numpy module in Python, which is a function that solves numerous linear algebra operations including linear systems, a topic approached in this work too.

Due to the high number of codes written in the chosen languages, some conventions and patterns were established:

- In order to reach statistical independence, five measurements were performed for each program, thus computing the mean and standard deviation
- Also, it is important to mention that we are considering four decimal places for each collected time
- Tables with the numerical data can be found in the annex section
- The computer was the same during all tests and, in addition, all other third-party programs were closed during run-time

It is worth specifying the resources used to obtain the elapsed time of all routines. In FORTRAN, we used a native environment function called **cpu_time**. To work with it, we must define two double precision variables: *start* and *finish*.The function is called twice in the code, in order to delimit the stretch we will analyze. Then, we display the elapsed time as the subtraction between *finish* and *start* variables. One consideration to make is regarding the information output format which corresponds to the *f6.5* format.

When dealing with Julia, we handled the **CPU_Time** package for measuring elapsed CPU time. At the beginning of the first code implementation, it took a lot of time finding this package due to the fact that it is not a Julia native module. Besides that, its usage is different from the methods applied to other programming languages regarding syntax. Basically, the time is computed by writing the following command in the stretch to be analyzed: *@time @CPUtime*.

During Python and Python + NUMBA implementation, we had to import the **time** module, and its usage is very similar to FORTRAN's. Essentially, what changes here is that once we start the counting we do not need to assign a finish variable. We simply do the *current time* minus *start* variable. Despite that, it is still possible to set up the same procedure written in FORTRAN using both variables.

Finally, considering all languages, MATLAB was the one with the easiest time calculation procedure. It only requires the **tic-toc** command, which is displayed, similarly to FORTRAN, delimiting the stretch routine on which the analyses will concentrate.

3.1 First Test

Assuming a random 3D matrix $(n,n,3)$, where n is the number of elements per matrix coordinate. Using the loop and vectored tests as described in the methodology, we were able to generate four bar plots, two for each test category and, inside each group, one for each n value.

3.1.1 Loop Operations

Figure 6 shows the time elapsed for each code, written in the specified programming languages, to be finished by using loop operations and by assigning 3000 as the number of elements in each matrix coordinate.

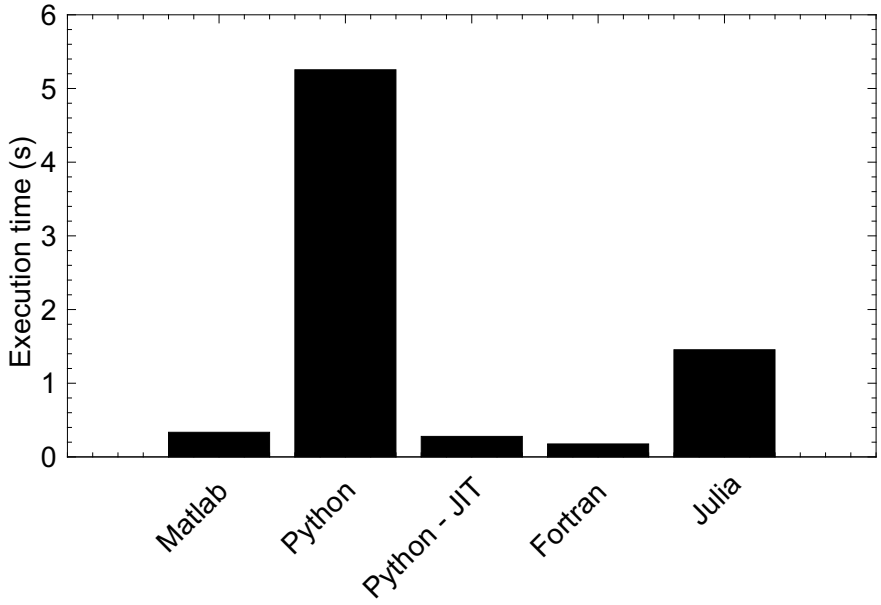


Figure 6: Execution time for Test 1 by applying loop operations, $n = 3000$ is the matrix coordinate elements number.

We note that among the languages studied, apparently Matlab, Python + JIT and

FORTRAN appear to have similar execution times. Python had the worst results by far, and Julia showed average conduct relative to the others. With this result, it is possible to speculate briefly how those programming languages will behave when we modify the n variable. Now, assigning 5000 as the new n value, we had Fig. 7 as a result:

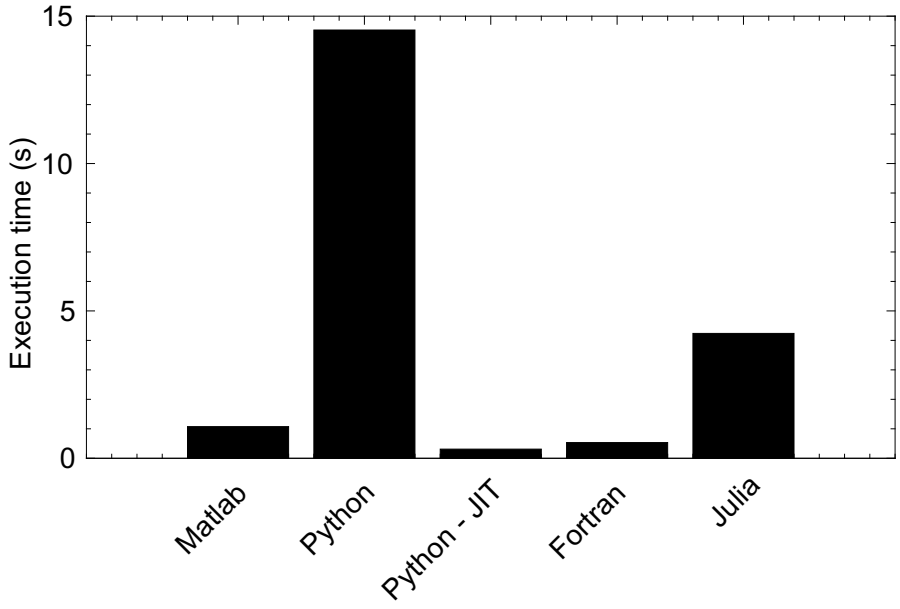


Figure 7: Execution time for Test 1 by applying loop operations, $n = 5000$ is the matrix coordinate elements number.

It is observed that our assumption was correct: there were only minor changes in the performance behavior of each language. Overall, the only noticeable variation now was the better performance of Python + JIT when compared to FORTRAN, although the differences between them seen negligible.

Interesting observations about both results are:

- In general, the fastest language solving loop operations for Test 1 could be either FORTRAN or Python + JIT.
- However the use of NUMBA module into Python programming allowed a shorter execution time than FORTRAN when n = 5000.
- The less efficient one was definitely standard Python, after all in both tests had the worst performance. We believe this is because it is an interpreted programming language
- Julia had an average performance showing the exact main purpose of this language, which is to be fast like C and easy to understand like Python

3.1.2 Vectored Operations

The following results are for intrinsic routines that use vectored operations. The execution time is seen in Fig. 8, where 3000 was allocated as the n value in the random matrix's x and y coordinates. It is possible to see that, as predicted, the time spent running vectored codes is way shorter than in the previous test.

Also noteworthy is this: despite the fact that the JIT module is used to speed up the operation, Python + JIT takes longer to execute than standard Python and also the other languages. This could mean that the NUMBA package is less effective when dealing with intrinsic routines than when dealing with loop operations.

Among the programming languages, FORTRAN was the one with best speed performance. Ideally, just like in the previous test, it is a foregone conclusion that this behavior will be repeated in the the round of tests.

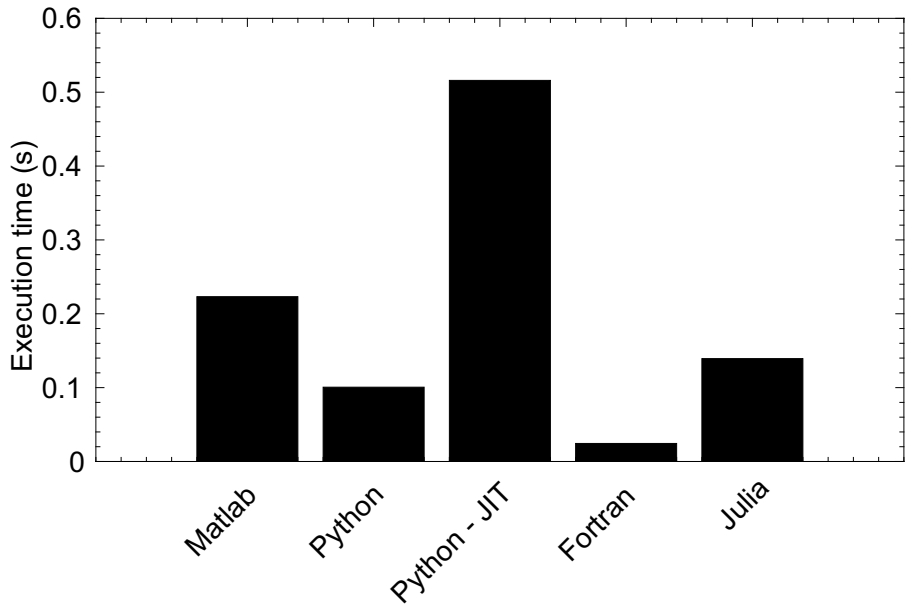


Figure 8: Execution time for Test 1 by applying vectored operations, $n = 3000$ is the matrix coordinate elements number.

An interesting fact found during the standard Python implementation of the code is that there are two ways of vectoring operations with this language: A[i] [j] [k] or A[i, j, k] as shown in Fig. 9. The unusual thing is that the last one is faster, than the first representation, to solve the same problem. We observed that this happened for different n numbers assigned.

The only modification in the codes scripts for the second run of tests is the substitution of the n variable for 5000, as seen in Fig. 10. The bar graphic format remained consistent with

the previous evaluation, as seen. Obviously, the amount of time spent here has changed. However there have been no improvements in the output of the programming languages as opposed to one another.

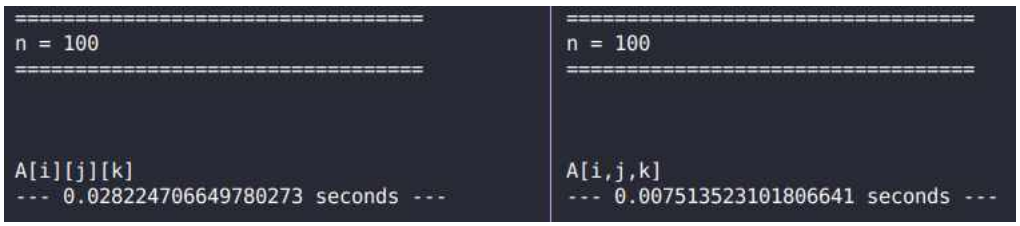


Figure 9: Comparative between elapsed time of two different python implementation

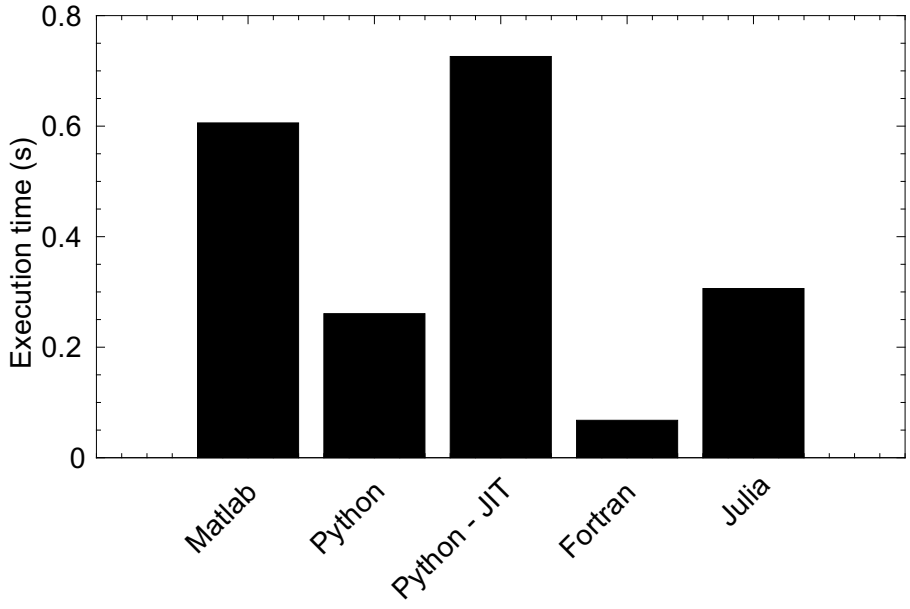


Figure 10: Execution time for Test 1 by applying vectored operations, $n = 5000$ is the matrix coordinate elements number.

These findings lead to some interesting conclusions, which are:

- In all cases, the fastest language solving vectored matrix issue is FORTRAN.
- An interesting fact about NUMBA is that its performance reduces by using vectored operations.
- Contrasting to the others, Matlab was the one which had the biggest time percentage increasing.
- Julia and standard Python still presents an average performance comparing with the others.

3.2 Second Test

Similarly to the first test, we also used here both loop and vectored approaches. However, the purpose of this problem was to test the multiplication of squared matrices (and each is *n*x*n* in dimension, with *n* being the number of elements per matrix coordinate). As normal, four bar plots were developed, two for each test group and, inside each group, one for each *n* number.

3.2.1 Loop Operations

Figure 11 shows the amount of time it took for each code, written in the specified programming languages, to be completed using loop operations and a *n* value of 250.

Matlab, Python + JIT, and FORTRAN appear to have similar execution times, much as they did in the previous loop assessment from Test 1. It is important to keep in mind that there was a huge time discrepancy between these languages and the others. By far the worst outcomes were achieved by Python, and Julia's behavior was average in comparison to the others.

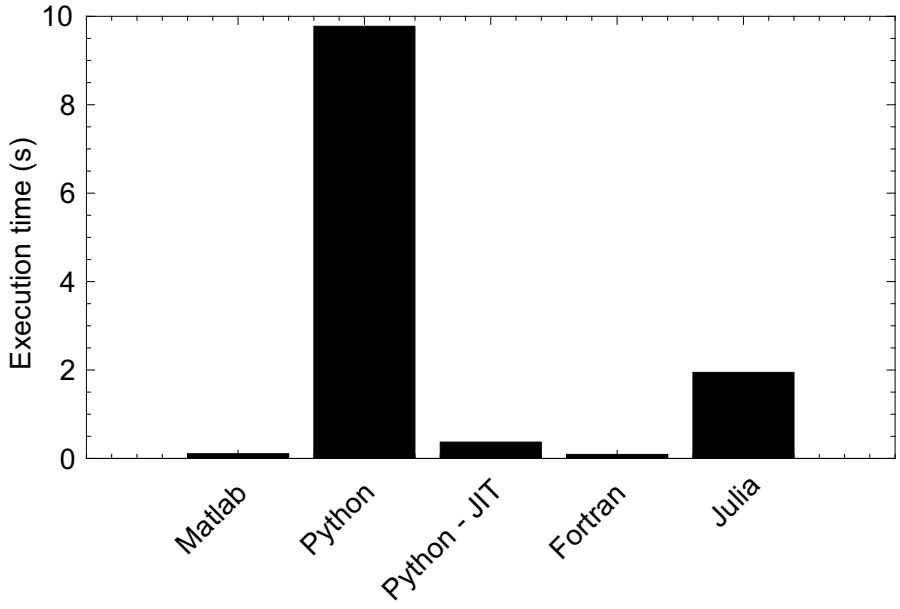


Figure 11: Execution time for Test 2 by applying loop operations, $n = 250$ is the square matrix order and it was assigned for all languages.

With this information, we can make some accurate predictions about how certain the chosen programming languages will react when the *n* variable is changed. With 350 as the new *n* value of matrices order, we now have Fig. 12 as a result. However, rather than showing all of the languages, we choose to only present the ones with the best performance after both Python and Julia substantially increased their time. For all languages performances consult the annex section.

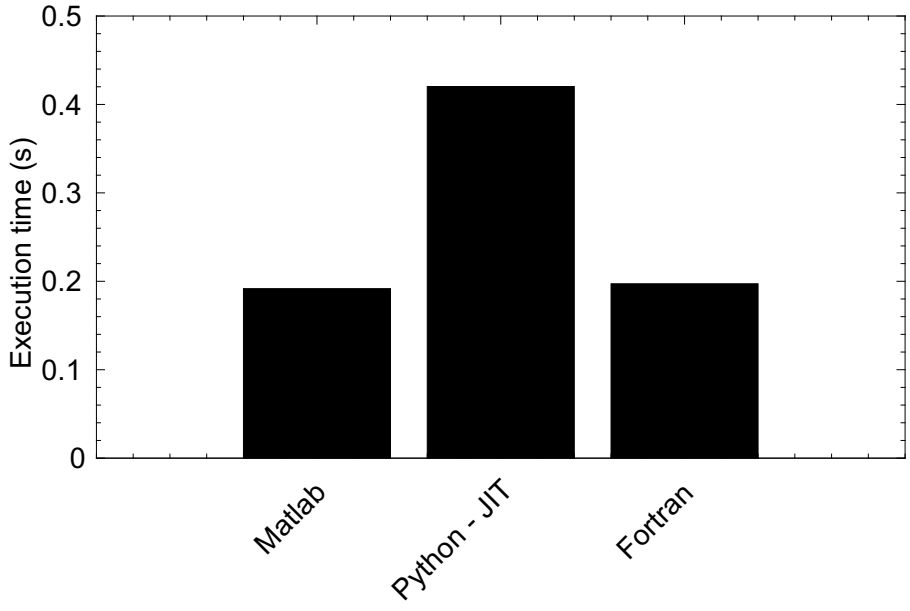


Figure 12: Execution time for Test 2 by applying loop operations, $n = 350$ is the square matrix order and it was assigned for Matlab, Python - JIT and FORTRAN.

Although a good visualization on a macro scale is very difficult, since three of the original languages are shown here, it is noted that Matlab and FORTRAN are the most effective in this test, while Python + JIT loses efficiency as the array order increases .

Interesting observations about both results are:

- In general, the fastest language solving loop operations for Test 2 could be either FORTRAN or Matlab.
- However the use of NUMBA module into Python programming allowed a shorter performed execution time.
- Similarly to the loop setup from Test 1, the less powerful one was undoubtedly regular Python; after all, it performed the worst in both experiments. This, we suppose, is due to the fact that it is an interpreted programming language.
- Also, Julia had an average output demonstrating the very main aim of this language, which is to be as swift as C and as simple to understand as Python.

3.2.2 Vectored Operations

The time each code took, written in the specified programming languages, to complete using vectored operations and a n value of 1500 is shown in Fig. 13. Since there is no intrinsic function in Python that vectorizes the matrix multiplication operation, the NumPy module and

the **matmul** function were used. It is worth noting that there are no Python + JIT results, which is due to NUMBA's unsupported use of the **numpy.matmul** feature.

We can observe that Matlab and FORTRAN seem to have similar execution times, as they had in the previous loop evaluation of Test 2. Again, the worst results were performed by Python, noting that there is a great time discrepancy between these languages and the others. In addition, Julia's behavior was average compared to the others.

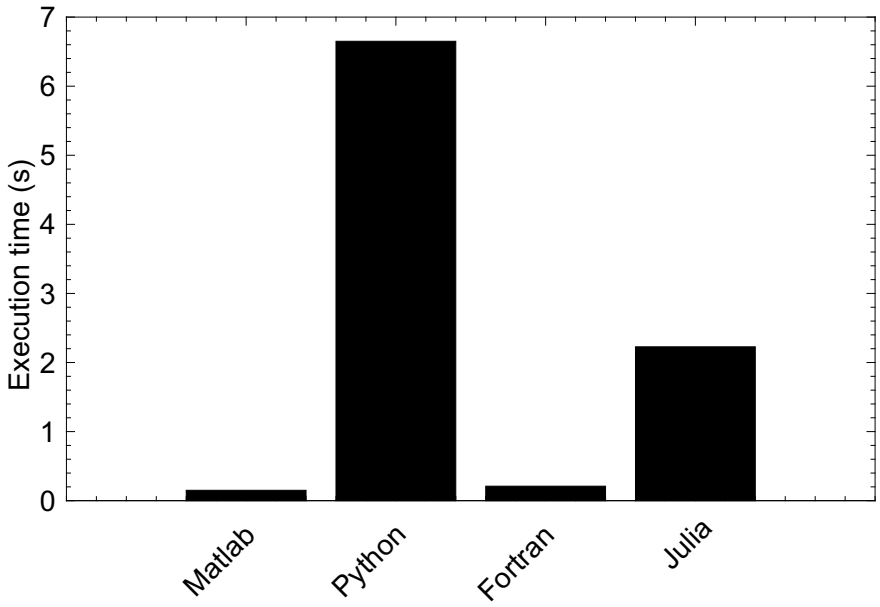


Figure 13: Execution time for Test 2 by applying vectored operations, $n = 1500$ is the square matrix order and it was assigned for all languages, with the exception of Python - JIT for which matrix multiplication modules are not supported.

We may make some good estimations based on this knowledge about how the chosen programming languages will behave when the n variable is updated. With 5000 as the new n value of matrices order, we now have Fig.14 as a result. However, just like in the previous test, instead of showing all the languages, we choose to present only those with the best performance after both Python and Julia significantly increased their time. For all four languages performance, please refer to the Annex section.

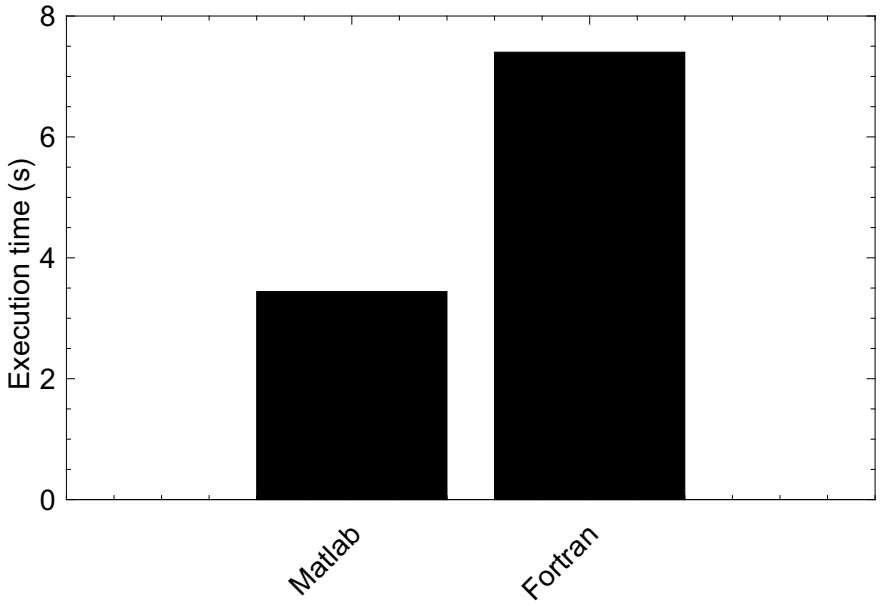


Figure 14: Execution time for Test 2 by applying vectored operations, $n = 5000$ is the square matrix order and it was assigned for Matlab and FORTRAN.

It is noted that for high orders, surprisingly, Matlab performs better than FORTRAN. That being known, it is possible to say that MATLAB is the most optimized language when running this test.

Interesting observations about both results are:

- Overall, the fastest language solving vectored operations for matrix multiplication is MATLAB, especially when assigning high orders.
- There is an incompatibility with the external module **numpy.matmul**, turning out to be unsupported with Python - JIT.
- The less efficient one, similar to the loop setup from Test 1, was undoubtedly standard Python; after all, it performed the worst in both experiments.
- Julia still presents an average performance comparing with the others.

3.3 Third Test

For the third test, unlike the first two, here we tried to evaluate the performance taking into account the execution time in iterative and recursive problems. The problem in question, as previously mentioned, is that given a certain index n, the machine should find the respective Fibonacci's sequence value. In addition, building on the previous tests, six bar graphs were developed here, three for each test group and, within each group, one per *n* index.

3.3.1 Iterative Operations

Figure 15 represents the execution time for the languages arranged for the iterative test. It was also assumed that the index would be 30, that is, when running the scripts, it was expected that we would find the sequence value for the respective number of desired iterations.

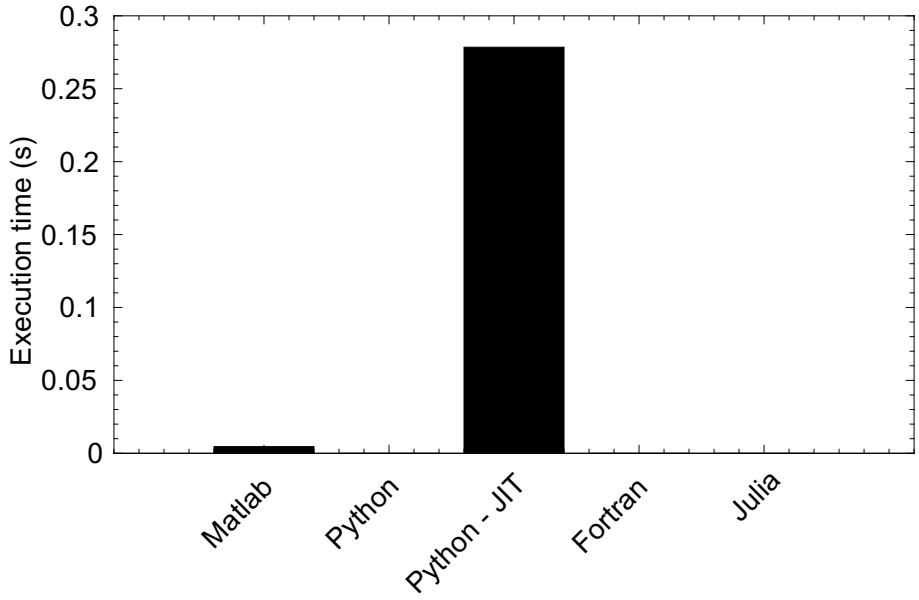


Figure 15: Execution time for Test 3 by applying iterative operations, $n = 30$ is the index series number for all languages.

We noticed a new interesting pattern here, among the languages studied: apparently for FORTRAN, Julia and standard Python, the execution time for the given problem is very small. In fact, for the number of decimal places initially stipulated for carrying out the tests, it was not possible to calculate them properly.

This implies that, in these languages, unlike previous tests that involved matrices, iterative processes in variables are extremely optimized. Tests were made assuming exorbitant values for the number of iterations and even so there were no significant changes. In general, it is also possible to observe in the figure that: curiously Python + JIT, was the least optimized programming language. Matlab also presented a slight execution time.

This problem is really intriguing, because unlike the previous ones, the hypothesis that the increase in the number of iterations would increase the execution time was wrong. As it is possible to analyze in Fig. 16, there was no significant changes, including in Python + JIT and Matlab, which leads us to believe that there might be some residue in the compiler and in the interpreter respectively. This fact further justifies the notion that, now, for all the programming languages analyzed, iterative processes that deals especially with variables are very fast and at the same time extremely optimized.

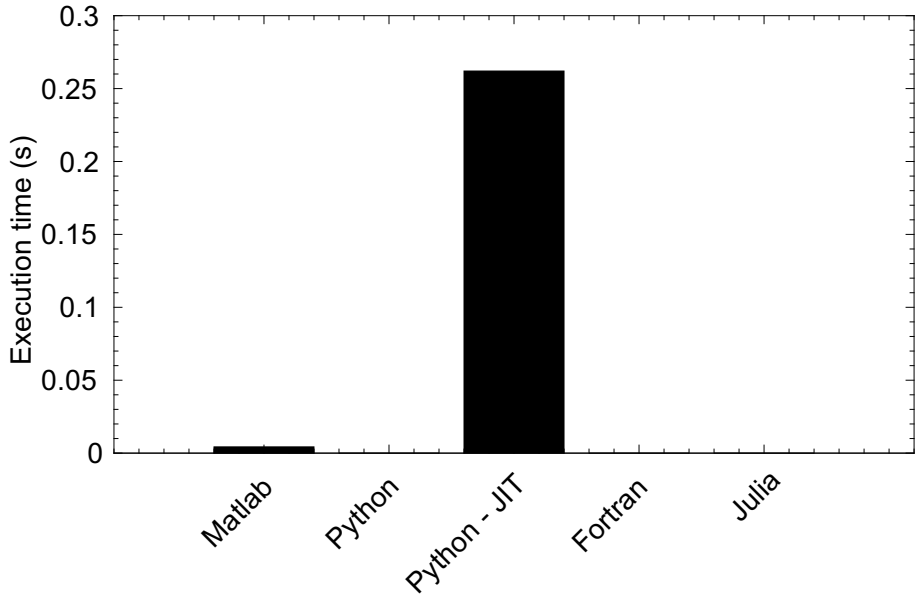


Figure 16: Execution time for Test 3 by applying iterative operations, $n = 45$ is the index series number for all languages.

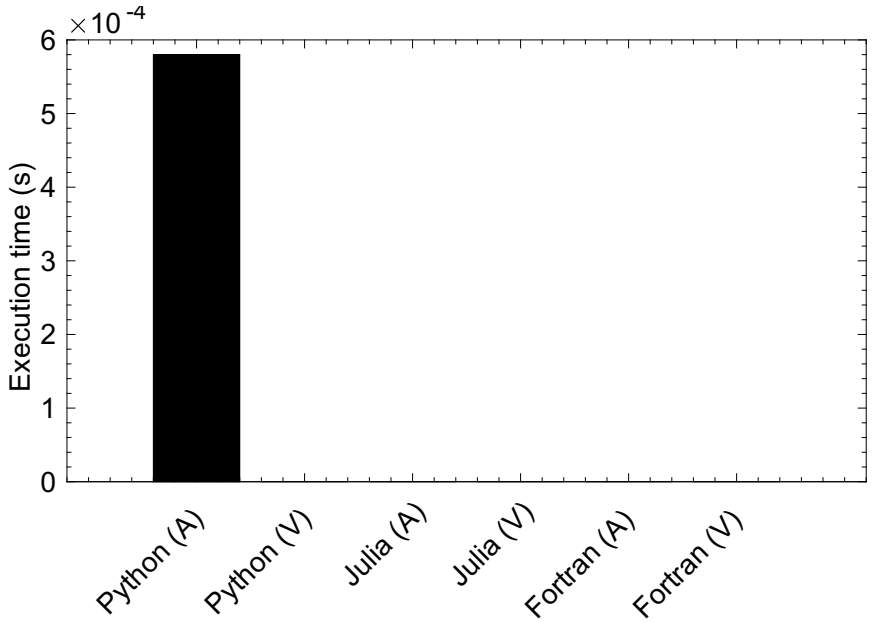


Figure 17: Comparative execution time for Test 3 by applying iterative variable and array operations, $n = 45$ is the index series number for Python, Julia and FORTRAN.

Finally, a test was also carried out in order to find the required number in the Fibonacci's sequence. However, now using a dynamic allocation vector, based on the user's input. As can be seen in Fig. 17, the number of iterations was also defined as 45. It can be seen from the comparison, set out above, that: in both Julia and FORTRAN, the two ways of implementing the

same problem are optimal. On the other hand, when it comes to allocating memory in a vector, it was expected that there would be an increase in the execution time, which in fact happened to Python.

3.3.2 Recursive Operations

For recursive procedures, the following results are given. Fig. 18 shows the execution times, where 30 are assigned as the n values for the calculated number index. It can be seen that the time spent on coding using recursive functions, as expected, is much longer than the preceding test.

Also noteworthy is the following: the languages of Matlab and Julia show a time of execution close to each other and also average when compared to the others. Among the programming languages, Julia was the one that presented the best speed performance by far, while the two types of Python are initially the slowest among them.

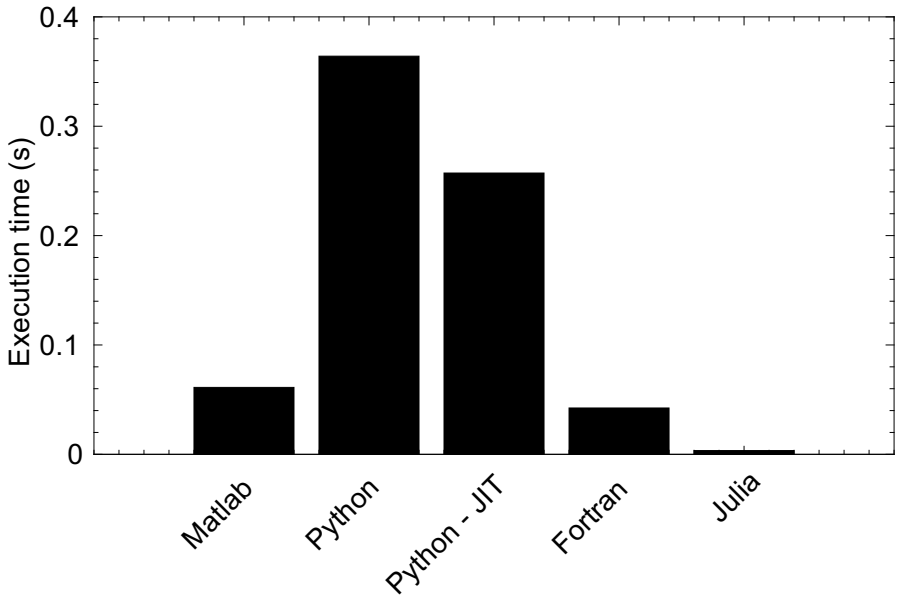


Figure 18: Execution time for Test 3 by applying recursive operations, $n = 30$ is the index series number for all languages.

However, with 38 as the new n value of number index, we now have Fig.19 as a result. It is possible to see that, unlike the previous figure, Python + JIT performed much better. It is also noted that, this same language obtained even better results when compared to Matlab, showing that in this case, when dealing with recursion, there is a loss of computational efficiency. The Fig. 20 shows an enlargement of Fig. 19 in order to get a better view of Julia and FORTRAN's performance.

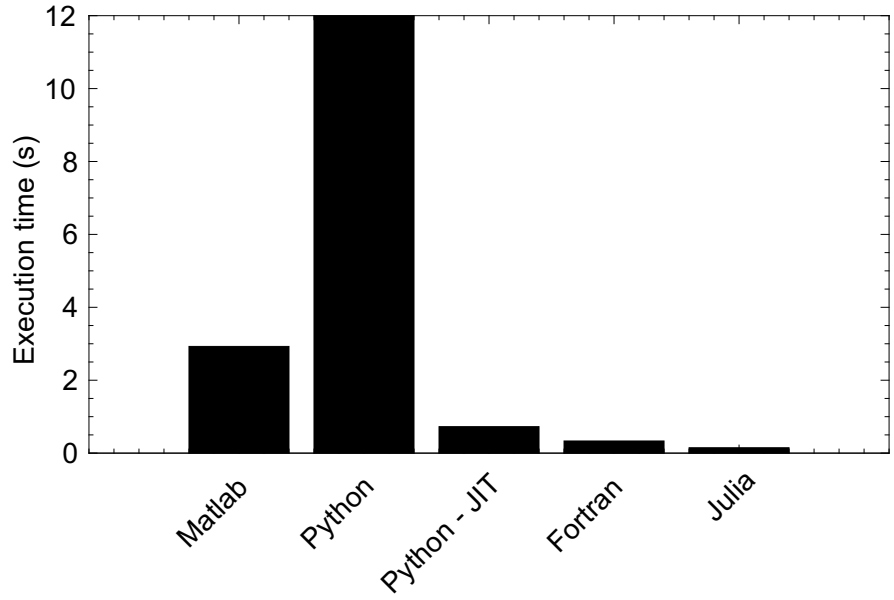


Figure 19: Execution time for Test 3 by applying recursive operations, $n = 38$ is the index series number for all languages.

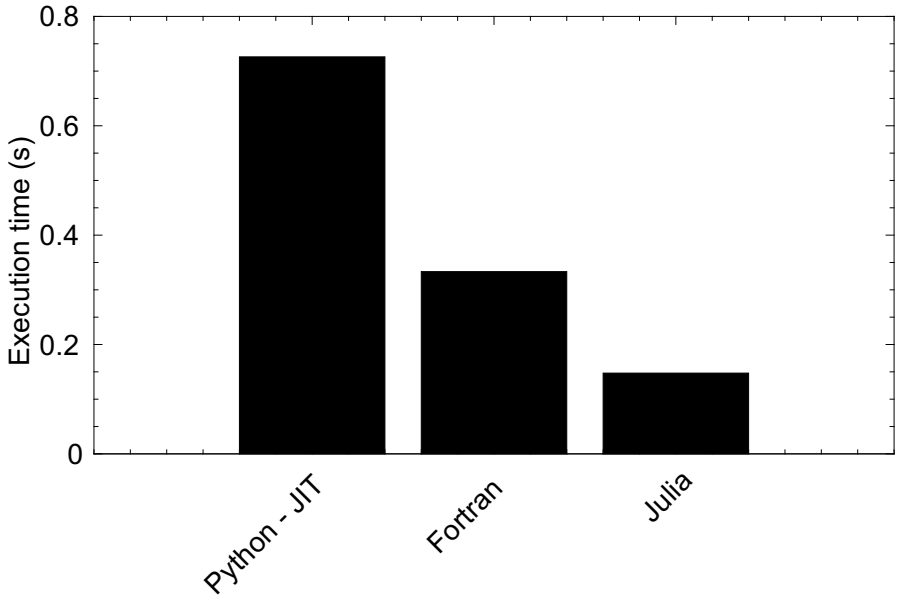


Figure 20: Execution time for Test 3 by applying recursive operations, $n = 38$ is the index series number for Python - JIT, FORTRAN and Julia.

One noteworthy finding is that Julia still performed very well even with index growth.

Interesting comments on both outcomes are:

- In both tests, Julia was the programming language with the best execution time, right

followed by FORTRAN

- The less efficient one was standard Python; after all, it performed the worst in both experiments.
- Python + JIT performed a better result over high orders indexes.

3.4 Fourth Test

For this test, as stated in the subsection 2.4, the goal here was to evaluate the elapsed time when writing an amount of data into a *.txt* file. Thus, using two arrays, where n equals to 10000 slots was the array dimension assigned for each, the results are shown in the Fig. 21.

It is important to mention that here, just like in the case from the sub-subsection 3.2.2, Python + JIT times are not displayed. Again, due to NUMBA's unsupported use of the **write** feature, so there are only results for Matlab, standard Pyhton, FORTRAN and Julia.

The results obtained surprisingly contradict the last tests: Python had the lowest processing power among the languages, followed for Fortan with an average result. Both Julia and Matlab had similar elapsed times.

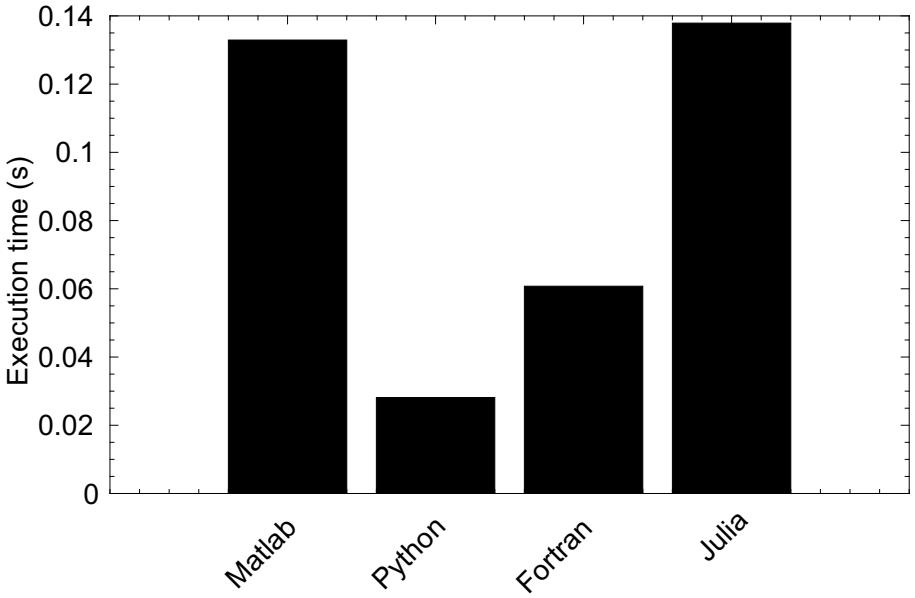


Figure 21: Execution time for Test 4, $n = 10000$ is the slots in each array for Python - JIT, FORTRAN, Julia and Matlab.

However, increasing 100 times n value, now corresponding to 1000000 slots for each array, we now have Fig. 22 as a result. It is possible to see that, unlike the previous figure, now

Julia language is the one with the best results. Both Python and FORTRAN also had some great performances.

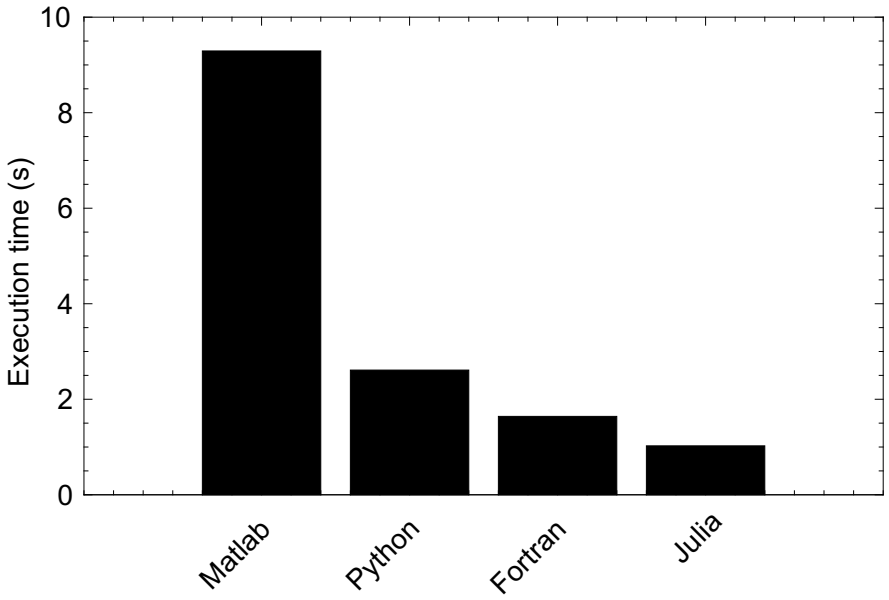


Figure 22: Execution time for Test 4, $n = 1000000$ is the slots in each array for Python - JIT, FORTRAN, Julia and Matlab.

Interesting comments on both outcomes are:

- In this test, for a considered small quantity of slots, Python was the programming language with the best execution time
- Increasing n value, Julia showed the best elapsed time, although it showed to be less efficient in the first test
- For both tests, FORTRAN showed good consistency
- The less efficient one was standard Matlab; after all, it performed the worst in both experiments
- Python + JIT is not compatible with the **write** module

3.5 Fifth Test

This is considered the most important test of this study since it takes into account most routines used in the previous tests. Our goal here was to evaluate the best programming language to solve the temperature distribution on a flat surface. A two-dimensional mesh was generated where each node corresponded to each n cell used. It is worth remembering that our objective

here is not to focus on the mathematical formalization of the problem, but rather to provide a short context and solve it.

Figure 23 shows the elapsed time each code to the linear system resolution, using n value of 64 nodes. Matlab, Julia and FORTRAN appear to have the best execution times. It is important to keep in mind that there was a huge time discrepancy between these languages and the two Python versions. Surprisingly, by far the worst outcomes were achieved by Python + JIT, which is an unexpected behavior.

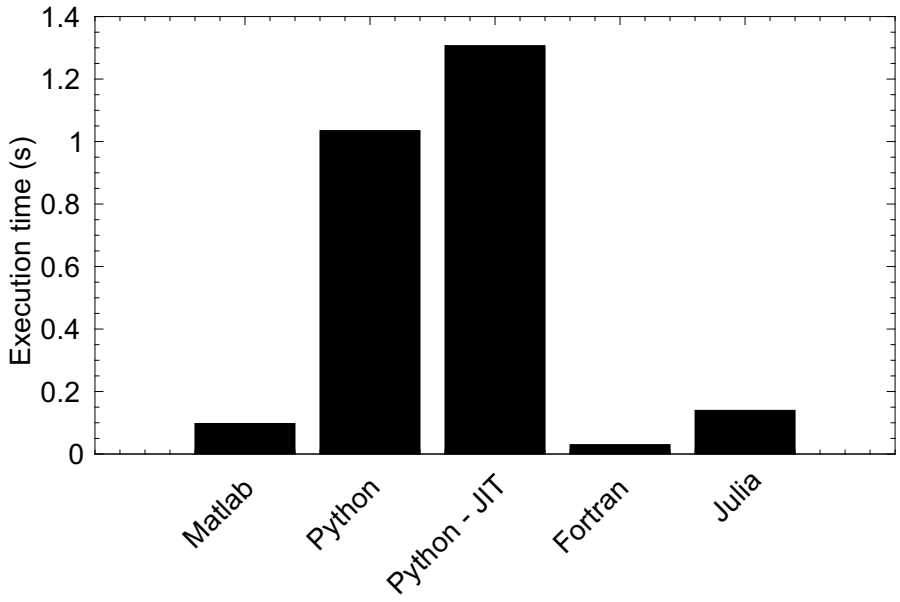


Figure 23: Execution time for Test 5 by applying loop operations, $n = 8x8$ is the mesh for all languages.

Now, assigning 144 as the new n value, we had Fig. 24 as a result. It is possible to note that the time scale has changed significantly, and there was a time increase for all programming languages displayed. However, some new findings can be observed here: when compared to the others, Python + JIT had a significant performance optimization, assuming lower execution time than Julia Language. This is at least curious, after all Julia is a language that already has the intrinsic JIT system. However, it is still compiled. On the other hand, Python + JIT continues to be interpreted. Apparently, as the number of iterations increases and the linear system to be solved becomes larger and more complex, the iterations performed in the codes seem to have a higher influence on overall computing time. This aspect modifies the time relationship between languages.

However, standard Python shows up as being the slowest in both tests as FORTRAN shows up to be the fastest among them. Because of that, as a prediction, it is mostly certain that

this situation will repeat itself in the next test.

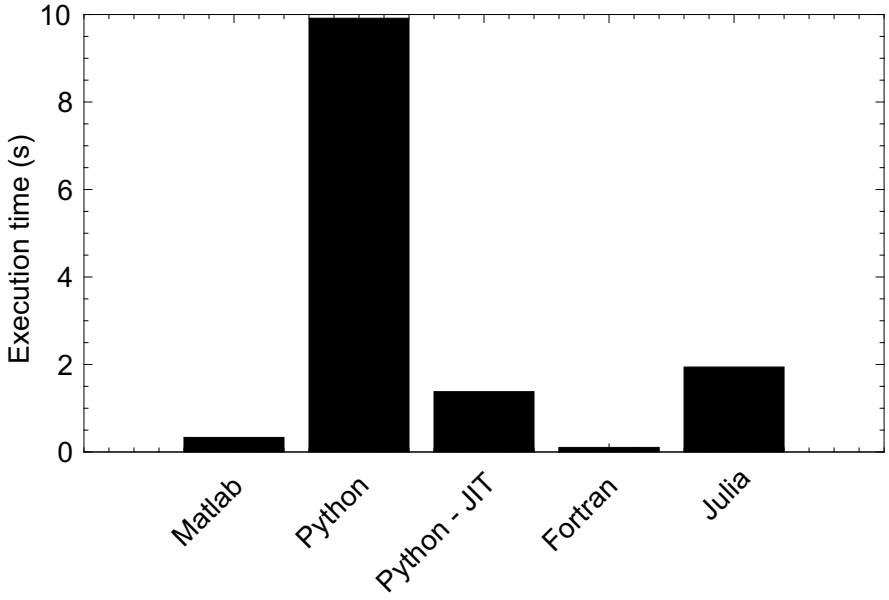


Figure 24: Execution time for Test 5 by applying loop operations, $n = 12x12$ is the mesh for all languages.

The results obtained for Fig. 25 are very similar in shape to the previous figure. Now, for n equal to 196, we can observe here that both Python + JIT and Matlab improved their performances when compared to the others, at least relatively.

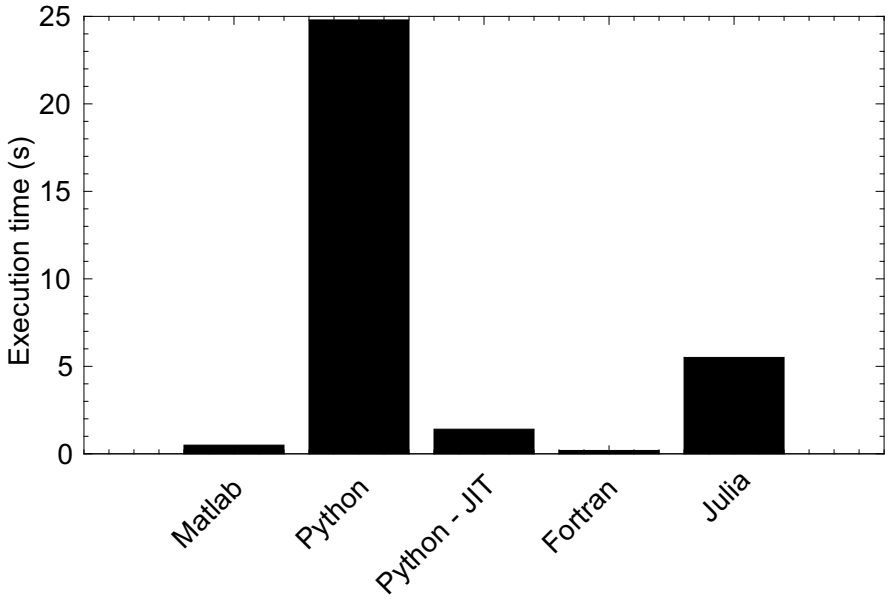


Figure 25: Execution time for Test 5 by applying loop operations, $n = 14x14$ is the mesh for all languages.

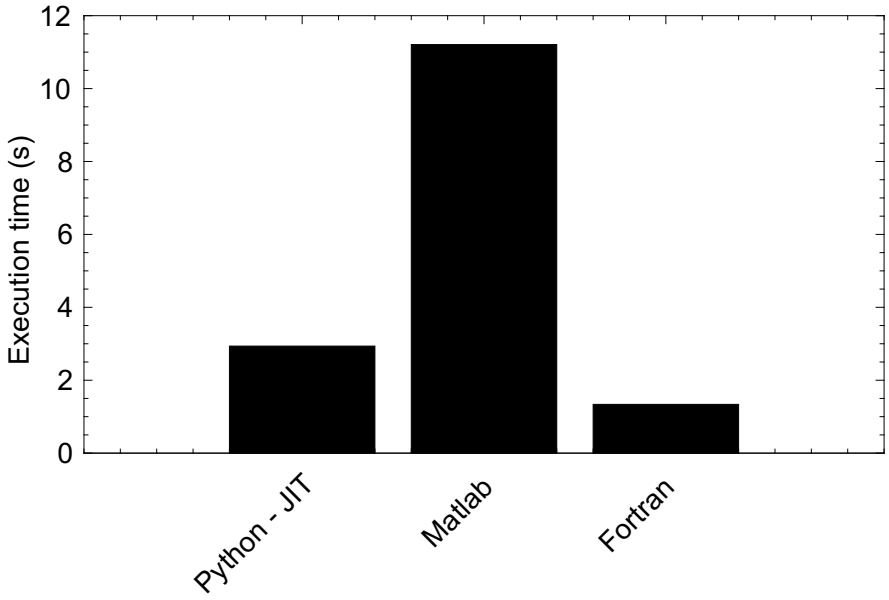


Figure 26: Execution time for Test 5 by applying loop operations, $n = 25 \times 25$ is the mesh for Python - JIT, Matlab and FORTRAN.

Due to the importance of this test, we wanted to simulate the execution time of the different languages taking into account more than just one mesh refinement. That is, with n equal to 64, 144 and 196. However, from the results obtained in Fig. 25, we realized that we could easily use a finer mesh based on the optimal performances. In the light of the above reasons, as seen in Fig. 26, the same procedure was carried out for such languages, although with the difference of a large increase in the number of elements of the mesh, making it much more refined than before.

These findings lead to some interesting conclusions, which are:

- For a considered small number of attributed nodes in the grid mesh, Python had a better execution time when compared to Python + JIT
- Increasing n value, Python + JIT showed a relative efficiency gain. When we used n as 625 nodes, it was even superior to Matlab
- During the four tests, FORTRAN showed a good consistency and the best performance among the programming languages
- The less efficient one was standard Python; after all, it performed the worst in all the experiments
- As usual, Julia showed an average elapsed time

Figures 27-34 show the temperature distribution obtained for both Matlab and Python (using the Matplotlib plot module). Each figure is related to one of the evaluated meshes during the run-time codes. It is possible to notice the different colors that represent the surface temperature.

Inside it, the temperature is distributed into values that are among those assigned on the walls by the Dirichlet's boundary conditions. Since the temperatures on the north and south sides of the square are the same, this corroborates to the creation of an axis of temperature symmetry that quotes the surface in the center from the left to the right face. It is also noted that qualitatively the result is very close to what was expected initially.

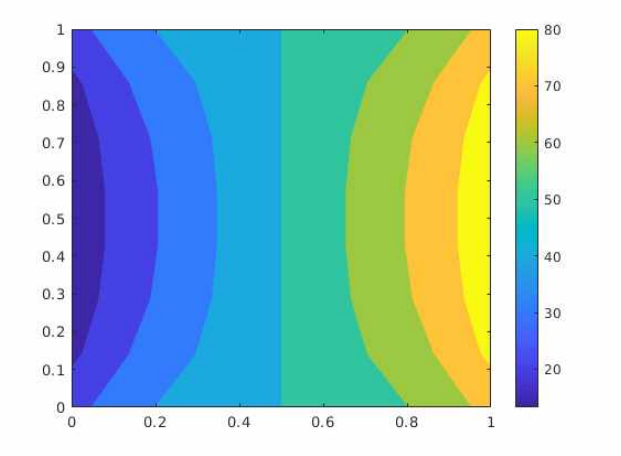


Figure 27: Matlab contour plot - Surface temperature distribution for steady regime, mesh - 8x8

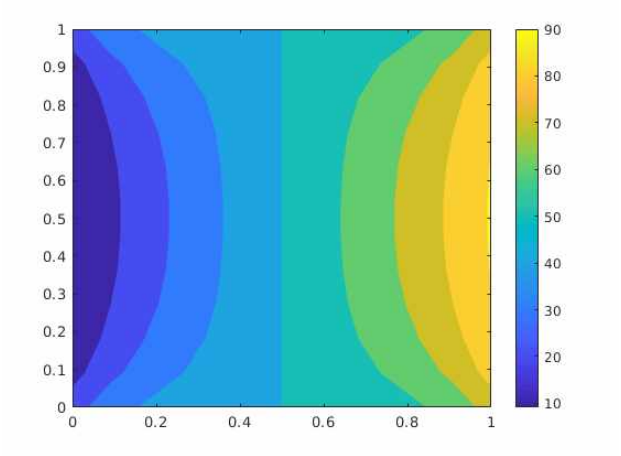


Figure 28: Matlab contour plot - Surface temperature distribution for steady regime, mesh - 12x12

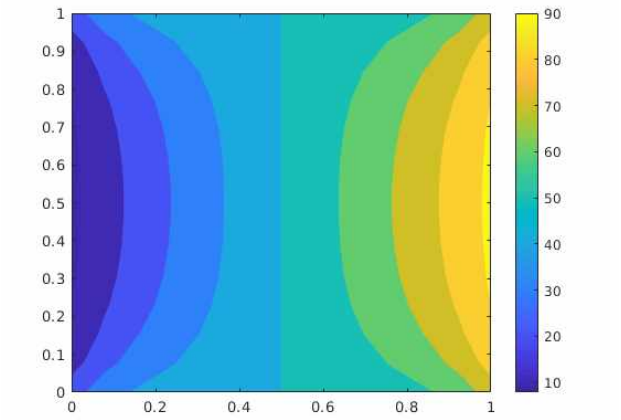


Figure 29: Matlab contour plot - Surface temperature distribution for steady regime, mesh - 14x14

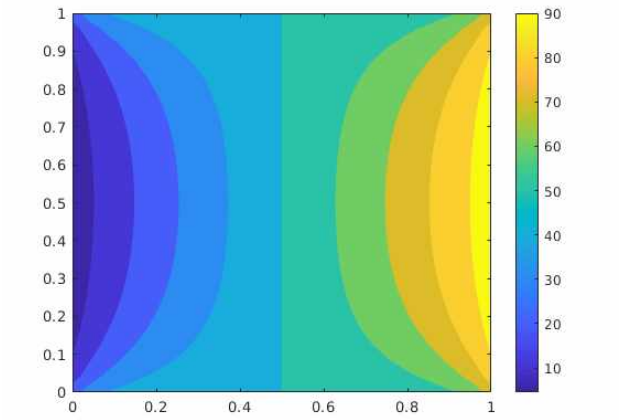


Figure 30: Matlab contour plot - Surface temperature distribution for steady regime, mesh - 25x25

In Figs. 27 and 31 seem to be very discrete, which is to be expected given the coarse mesh used. However, as we refine the mesh, that is, increase the number of stitches in it, the result is considerably improved, as shown in the subsequent figures. So, it can be seen a much smoother transition can be seen.

We did not evaluate the processing error of the solution. However, it is known that the error of 25x25 when compared to 8x8 is considerably smaller. To find the processing error, we should know the real solution related to the problem and later make a comparison between them.

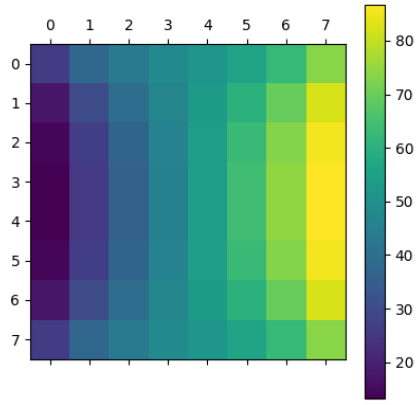


Figure 31: Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 8x8

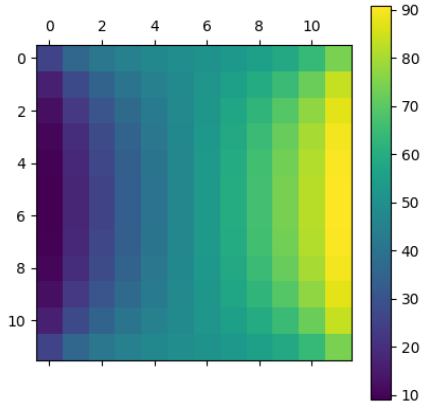


Figure 32: Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 12x12

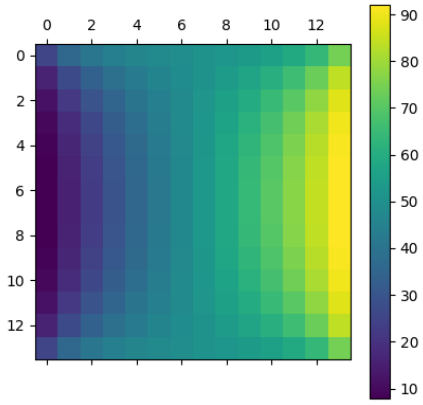


Figure 33: Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 14x14

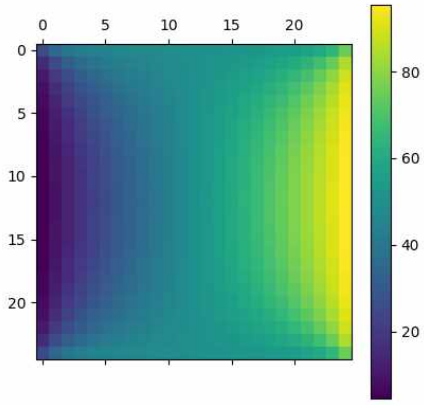


Figure 34: Matplotlib matrix plot - Surface temperature distribution for steady regime, mesh - 25x25

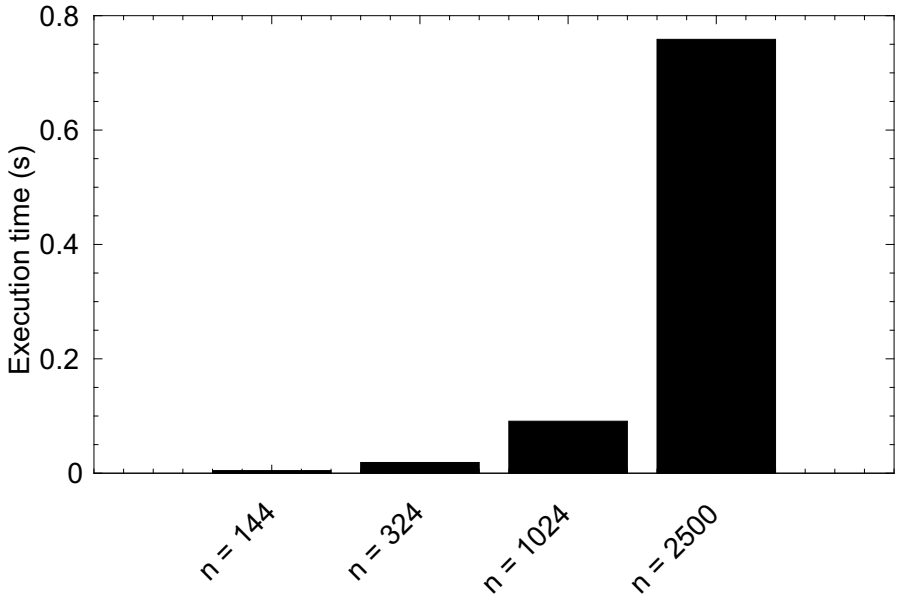


Figure 35: Example of execution time for Python using linalg/LAPACK method, various n values.

A final test was performed, though only as a comparison with the previous ones. Looking at the above observations, we deepened our analysis in the Python standard, since it presented the worst execution times of all languages. So, not differently, here we tried to evaluate the computational efficiency of this language, with the difference that we would now use ready-made routines and extremely more optimized for the resolution of the linear system. Specifically, in relation to this programming language, the linalg/LAPACK library was used.

The result, as shown in Fig. 35 is not surprising at all. It was expected that intrinsic or third-party routines would have better results. However, because there are infinite possibilities of solving this same problem using the third-party routines, the test was carried out taking into account the continuous refinement of the mesh in the following sequence: 144, 324, 1024 and finally 2500.

Two preliminary conclusions are reached here: the first one refers, as usual, to the fact that routines already completed and previously implemented in CFD codes are useful and vital for the performance of a given code in view of its execution time. In addition, as it is possible to observe in Fig. 35, with the mesh refinement the computational execution time increases exponentially. In view of this, it is essential to use the most optimized structures to facilitate the processes and problem-solving of this segment of computational numerical dynamics.

3.6 Overall results

Figure 36 shows the Matrix of times found during some of the tests performed. With this matrix it is possible to have a better idea as to the performance of each language in a standardized way. That is, the longest time spent by a language is admitted to be 1 or 100%, so we will have percentages of other languages based on this classification.

White represents when it was not possible to execute the code, as is the case of functions not intrinsic to Python when dealing with Numba. The dark blue color represents those with the shortest execution time.

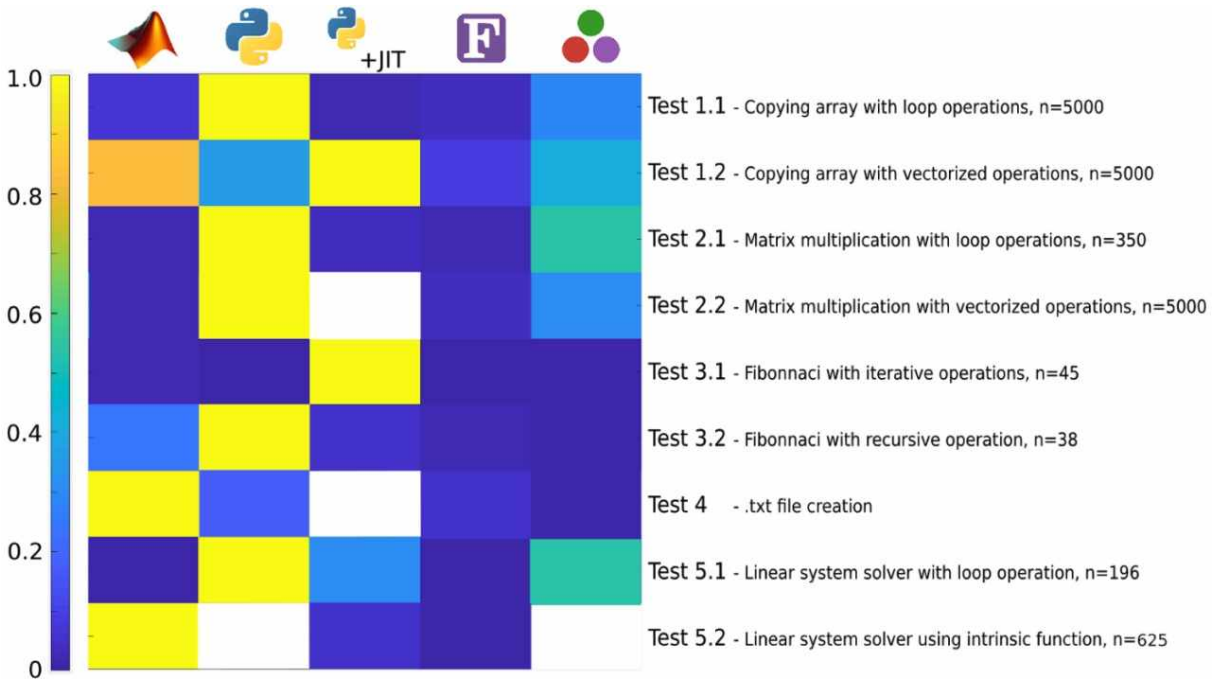


Figure 36: Matrix of standardized elapsed times, where as the colors become darker, it indicates that there is the shortest run-time

Those results are in concordance with Kouatchou [2019] findings, which is one primal reference in relation to elapsed time comparison. From the matrix it is possible to see that among the tests described here, those with the most darkest blue dots are Matlab and FORTRAN. It is also possible to find out that Julia, as stated above, performed in an average way in relation to all the tests.

During the coding process, a brief summary of the user's impression of each programming language are listed below:

- Julia has a scarce documentation, making difficult to implement some of the routines,

especially with variable scope and time measure

- Fortran is widely embraced by the scientific computing community, so there are extensive documentation about this subject
- MATLAB, commonly widespread among engineers and it has many discussion forums
- Python is considered a high order language with a very extensive documentation and caring dev community. The same applies to Numba module. Among the tested languages is the one with fastest learning curve

4 Conclusion and Further Work

By analysing the results, it is possible to reach some first conclusions about the usage and performance of programming languages solving the proposed problems.

The tests presented here have a bias based solely on the elapsed time of a code execution process in a given language, as mentioned in the Introduction. However, it is also important to note that during the writing of the scripts, the syntax of each language was checked, as well as the difficulty in finding a specific function or information with the language community.

In view of the available documentation, Julia was the one which had the worst impression among the languages. Despite the existence of the Julia community, it is not well developed. During the creation of the codes, in particular to compute the performance of the language, it has been quite extensive since then, due to the scarce documentation. However, it can be considered as a language with a lot of potential, as it is fast and easy to understand. After all, just like Python, it is classified as high-level language. In addition, the language community tends to grow more and more. Regarding computational time, it is possible to conclude that Julia has a medium performance when compared to the others, despite the fact that is the most efficient one in the recursion and the Fourth test. That said, the general conclusion about this language is that although it is quite innovative and simple to use, the fact that it does not have an active community, still in full development, plays a role against the language. For this reason, at first we believe that among the languages used, Julia does not prove to be the best option for an engineer at the beginning of his studies in CFD and scientific computing.

FORTRAN, as a language that has been widely embraced by scientific computing community, has extensive documentation about this subject. This is beneficial since the syntax is considered to be of low-level and is much more complex than the previous one. During the implementation of the codes, our big concern relied on the format of the output's results. However, in relation to computational cost time, it is fast in vectored and non-vectored forms, in addition to being quite optimized in terms of recursion as well as Julia. For this reason, in view of a good documentation and a superior execution time for most tests, we consider FORTRAN as an excellent choice for those who want to venture into this area of knowledge.

As for Matlab, perhaps the most commonly used language by engineers, it has comprehensive documentation as well as an active forum where questions are easily answered by members of the community. Furthermore, because it is a C++ library, even though it is a low-level language like FORTRAN, we believe that its widespread use will not be a problem for novice programmers who want to start with this language. In terms of computational time, the language is fast for the vast majority of tests, making its application quite interesting, especially

in scientific computing, as in the last test.

With regard to Python's language, the documentation for this programming language is extensive, and the community is very involved, with new libraries being added on a regular basis. When it comes to looping, the simplistic Numba package improves Python's performance, but when it comes to vectoring, it has the opposite effect. Furthermore, it should be noted that there are certain drawbacks in relation to the use of functions that are not exclusively native to Python, even in relation to Numba. When we tried to use any of the Numpy library's features alongside with Numba, we discovered that the computer could not interpret the code using this module.

Python is the slowest language of those studied in terms of computation time (perhaps due to the fact that it is interpreted). However, using Numba or more optimized methods and routines, as in Test 5, we were able to achieve truly amazing results for this language, even outperforming FORTRAN. As a result, we believe it has the most potential among those with prior experience.

Considering all the tests, we have found that the best core languages among those studied are FORTRAN and Matlab. This is due to accessible information, a strong community on Internet forums and, mainly, the fact that they performed better in the face of most tests. At first, it would be reasonable to select FORTRAN as the best language since among the most optimized, it is the one that presents the best results. It is easy to see why this programming language is commonly picked by a vast number of CFD groups all over the world.

However, in the face of another point of view, we could predict that soon Python is going to perform well. This is due to the fact that:

- Language development is still in full swing
- New optimized routines are being developed all the time
- Its use is free and it is not necessary to have a license.

In the near future, this could allow much more efficient routines, far better than those found in all the other programming languages.

Finally, it is possible to conclude that: for simple tests, where there is no need for computational performance and the number of analyzes is considered small, Python may be the best option due to its easy syntax and the fact that it is largely embraced by the community. At the same time, when dealing with more sophisticated problems/projects, it is preferable to use Fortran. This is because of the faster runtime when compared to the other programming languages, not to mention its widespread use by scientific computing community.

For further work, we believe that the main focus should be on implementing new

tests, which are even richer and more sophisticated. It would also be interesting to conduct parallelization tests, bring additional computational languages to the analysis and making a comparison between those routines when executed using CPU and GPU. Also, we seek to check whether the results are similar as well as the general conclusion is maintained.

References

Z. Anik and F. Baykoç. Comparison of the most popular object-oriented software languages and criterions for introductory programming courses with analytic network process: A pilot study. *Computer Applications in Engineering Education*, 19(1):89–96, 2011.

A. Backes. *Linguagem C - completa e descomplicada*. Elsevier, Rio de Janeiro, 1st edition, 2013.

R. L. Burden and D. J. Faires. *Análise numérica*. Cengage Learning, São Paulo, 3rd edition, 2015.

I. Danaila. *An introduction to scientific computing : twelve computational projects solved with MATLAB*. Springer, New York, 2007. ISBN 978-0-387-30889-0.

J. Domke. Julia, Matlab and C. https://justindomke.wordpress.com/2012/09/17/julia-matlab-and-c/, 2020.

H. Fehr and F. Kindermann. *Introduction to Computational Economics Using Fortran*. Oxford University Press, 1st edition, 2018. ISBN 9780198804390.

J. H. Ferziger. *Computational methods for fluid dynamics*. Springer, 4th edition, 2020. ISBN 9783319996912,9783319996936.

J. V. Guttag. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. The MIT Press, 2nd edition, 2016. ISBN 9780262529624.

JuliaLang Organization. Julia 1.6 documentation. <https://docs.julialang.org/en/v1/>, March 2021.

J. Kouatchou. Basic Comparison of Python, Julia, Matlab, IDL and Java (2019 Edition). *NASA*, 2019. doi: 10.5281/zenodo.3675797.

A. V. Mohanan, C. Bonamy, M. C. Linares, and P. Augier. FluidSim: Modular, Object-Oriented Python Package for High-Performance CFD Simulations. *Journal of Open Research Software*, 7, 2019. doi: 10.5334/jors.239.

Numba Pydata Organization. Stable Numba. <https://numba.readthedocs.io/en/stable/user/jit.html>, March 2021.

S. Pawar and O. San. CFD julia: A learning module structuring an introductory course on computational fluid dynamics. *Fluids*, 4(3):159, Aug. 2019. doi: 10.3390/fluids4030159.

R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Oct. 2017. doi: 10.1145/3136014.3136031.

Precise Simulation. CFDTool - MATLAB CFD Simulation GUI Toolbox. <https://github.com/precise-simulation/cfdtool/releases/tag/1.7.1.1>, Retrieved June 22 2021.

S. Raschka. Numeric matrix manipulation - the cheat sheet for Matlab, Python Numpy, R and Julia. http://sebastianraschka.com/Articles/2014_matrix_cheatsheet.html, 2014.

F. M. White. *Viscous Fluid Flow*. McGraw-Hill,Inc, Pennsylvania, 2nd edition, 1991.

Annex

Table 1: First Test - Comparison Loop Test for n = 3000

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.3344	0.0091	0.3286	0.3313	0.3503	0.3289	0.3331
Python	5.2540	0.1037	5.4153	5.1255	5.2380	5.2538	5.2376
Python - JIT	0.2785	0.0052	0.2801	0.2815	0.2795	0.2694	0.2819
FORTRAN	0.1762	0.0025	0.1727	0.1744	0.1782	0.1784	0.1771
Julia	1.4544	0.0057	1.4595	1.4560	1.4597	1.4478	1.4491

Table 2: First Test - Comparison Loop Test for n = 5000

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	1.0743	0.0437	1.0157	1.1085	1.1191	1.0845	1.0439
Python	14.5304	0.3399	14.5490	14.0690	14.4128	14.6133	15.0081
Python - JIT	0.3110	0.0155	0.3000	0.3277	0.3249	0.3105	0.2917
FORTRAN	0.5334	0.0084	0.5252	0.5261	0.5328	0.5454	0.5373
Julia	4.2368	0.0566	4.2589	4.1407	4.2850	4.2648	4.2345

Table 3: First Test - Comparison Vectored Test for n = 3000

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.2232	0.0064	0.2226	0.2303	0.2242	0.2258	0.2129
Python	0.1006	0.0007	0.1003	0.1010	0.0998	0.1016	0.1003
Python - JIT	0.5160	0.0330	0.5750	0.5028	0.5020	0.5014	0.4988
FORTRAN	0.0244	0.0006	0.0236	0.0244	0.0243	0.0245	0.0253
Julia	0.1393	0.0005	0.1395	0.1400	0.1394	0.1387	0.1389

Table 4: First Test - Comparison Vectored Test for n = 5000

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.6058	0.0135	0.5858	0.6180	0.6001	0.6180	0.6073
Python	0.2607	0.0073	0.2498	0.2594	0.2648	0.2602	0.2693
Python - JIT	0.7261	0.0016	0.7277	0.7262	0.7273	0.7237	0.7254
FORTRAN	0.0678	0.0021	0.0656	0.0710	0.0672	0.0667	0.0683
Julia	0.3062	0.0007	0.3065	0.3053	0.3071	0.3057	0.3064

Table 5: Second Test - Comparison Loop Test for n = 250

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.1063	0.0045	0.1055	0.1109	0.1016	0.1024	0.1109
Python	9.7752	0.2962	9.6682	10.2123	9.4851	9.5771	9.9331
Python - JIT	0.3695	0.0138	0.3805	0.3499	0.3702	0.3840	0.3631
FORTRAN	0.0908	0.0065	0.0812	0.0951	0.0960	0.0948	0.0868
Julia	1.9467	0.0179	1.9321	1.9775	1.9382	1.9458	1.9398

Table 6: Second Test - Comparison Loop Test for n = 350, Python n = 250

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.1917	0.0093	0.1824	0.1989	0.1905	0.2034	0.1832
Python	9.7752	0.2962	9.6682	10.2123	9.4851	9.5771	9.9331
Python - JIT	0.4203	0.0087	0.4241	0.4235	0.4258	0.4231	0.4048
FORTRAN	0.1972	0.0073	0.1980	0.2016	0.2054	0.1948	0.1862
Julia	5.3298	0.0354	5.3309	5.2869	5.3838	5.3142	5.3331

Table 7: Second Test - Comparison Vectored Test for n = 1500

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.1504	0.0283	0.1067	0.1377	0.1674	0.1752	0.1652
Python	6.6500	0.1553	6.5513	6.6527	6.9153	6.5972	6.5335
Python - JIT	-	-	-	-	-	-	-
FORTRAN	0.2100	0.0097	0.2018	0.2073	0.2263	0.2043	0.2101
Julia	2.2273	0.2129	2.5943	2.0863	2.2263	2.1425	2.0872

Table 8: Second Test - Comparison Vectored Test for n = 5000, Python and Julia n = 1500

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	3.4418	0.0829	3.3145	3.4067	3.5079	3.5130	3.4668
Python	6.6500	0.1553	6.5513	6.6527	6.9153	6.5972	6.5335
Python - JIT	-	-	-	-	-	-	-
FORTRAN	7.4019	0.0777	7.3121	7.3443	7.3951	7.4990	7.4592
Julia	2.2273	0.2129	2.5943	2.0863	2.2263	2.1425	2.0872

Table 9: Third Test - Comparison Iterative Test for n = 30

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.0046	0.0034	0.0084	0.0067	0.0057	0.0006	0.0014
Python	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Python - JIT	0.2785	0.0052	0.2801	0.2815	0.2795	0.2694	0.2819
FORTRAN	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Julia	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 10: Third Test - Comparison Iterative Test for n = 45

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.0043	0.0037	0.0087	0.0035	0.0076	0.0012	0.0004
Python	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Python - JIT	0.2620	0.0324	0.2314	0.3015	0.2925	0.2417	0.2431
FORTRAN	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Julia	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 11: Third Test - Comparison Between Variable and Iterative Array for n = 45

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Python(A)	0.0006	0.0001	0.0006	0.0007	0.0005	0.0004	0.0007
Python(V)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Julia(A)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Julia(V)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
FORTRAN(A)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
FORTRAN(V)	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Table 12: Third Test - Comparison Recursive Test for n = 30

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.0612	0.0155	0.0876	0.0524	0.0538	0.0500	0.0622
Python	0.3641	0.0109	0.3605	0.3744	0.3750	0.3490	0.3615
Python - JIT	0.2572	0.0416	0.2772	0.2864	0.2976	0.2171	0.2078
FORTRAN	0.0424	0.0047	0.0491	0.0442	0.0397	0.0422	0.0367
Julia	0.0034	0.0003	0.0033	0.0039	0.0034	0.0033	0.0033

Table 13: Third Test - Comparison Recursive Test for n = 38

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	2.9271	0.0491	2.8805	2.8772	2.9835	2.9694	2.9251
Python	11.9996	0.3954	11.8547	12.6795	11.6455	11.8984	11.9197
Python - JIT	0.7261	0.0016	0.7277	0.7262	0.7273	0.7237	0.7254
FORTRAN	0.3333	0.0315	0.3124	0.3507	0.3531	0.2884	0.3618
Julia	0.1477	0.0019	0.1475	0.1471	0.1462	0.1509	0.1468

Table 14: Fourth Test - Comparison for n = 10000

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.1329	0.0082	0.1351	0.1186	0.1345	0.1388	0.1376
Python	0.0281	0.0032	0.0295	0.0253	0.0255	0.0330	0.0274
Python - JIT	-	-	-	-	-	-	-
FORTRAN	0.0608	0.0488	0.0411	0.0317	0.1476	0.0439	0.0395
Julia	0.1379	0.0096	0.1509	0.1435	0.1331	0.1259	0.1360

Table 15: Fourth Test - Comparison for n = 1000000

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	9.2928	0.1638	9.5051	9.0524	9.3275	9.3294	9.2497
Python	2.6111	0.0653	2.5740	2.5276	2.6141	2.6403	2.6996
Python - JIT	-	-	-	-	-	-	-
FORTRAN	1.6415	0.0399	1.5905	1.6930	1.6167	1.6636	1.6438
Julia	1.0247	0.0457	1.0463	1.0743	1.0494	0.9874	0.9662

Table 16: Fifth Test - Comparison Loop Test for n = 8x8

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.0977	0.0467	0.0571	0.0746	0.0936	0.1776	0.0858
Python	1.0350	0.0454	0.9542	1.0605	1.0512	1.0593	1.0499
Python - JIT	1.3070	0.0083	1.3018	1.3019	1.3022	1.3210	1.3082
FORTRAN	0.0300	0.0007	0.0306	0.0305	0.0299	0.0301	0.0289
Julia	0.1398	0.0084	0.1382	0.1334	0.1545	0.1368	0.1362

Table 17: Fifth Test - Comparison Loop Test for n = 12x12

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.3335	0.0332	0.3142	0.3076	0.3512	0.3108	0.3839
Python	9.9160	0.3236	9.7246	9.8070	10.4672	9.6583	9.9230
Python - JIT	1.3788	0.0150	1.3842	1.3724	1.3841	1.3965	1.3566
FORTRAN	0.1047	0.0154	0.0922	0.0975	0.0985	0.1039	0.1312
Julia	1.9417	0.0404	1.8891	1.9864	1.9546	1.9672	1.9111

Table 18: Fifth Test - Comparison Loop Test for n = 14x14

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Matlab	0.4973	0.0240	0.5355	0.4800	0.5043	0.4758	0.4910
Python	24.7964	0.2083	24.7646	25.0032	24.6871	24.5231	25.0042
Python - JIT	1.4086	0.0277	1.4112	1.4253	1.4218	1.4246	1.3600
FORTRAN	0.1925	0.0096	0.1756	0.1985	0.1937	0.1976	0.1973
Julia	5.5088	0.1070	5.5332	5.6594	5.5000	5.4917	5.3599

Table 19: Fifth Test - Comparison of Matlab, Python - JIT and FORTRAN for n = 25x25

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
Python - JIT	2.9361	0.1464	2.7166	2.9634	2.8802	3.0985	3.0217
Matlab	11.2078	0.1227	11.2332	11.2183	11.0793	11.1150	11.3932
FORTRAN	1.3377	0.0055	1.3311	1.3345	1.3382	1.3459	1.3387

Table 20: Example of execution time for Python using linalg/LAPACK method, various n values

	t_m(t)	Dev(t)	x1	x2	x3	x4	x5
n=144	0.0046	0.0018	0.0067	0.0019	0.0043	0.0045	0.0055
n=324	0.0188	0.0043	0.0259	0.0175	0.0178	0.0144	0.0182
n=1024	0.0909	0.0071	0.0917	0.0883	0.0872	0.1028	0.0847
n=2500	0.7586	0.0215	0.7408	0.7743	0.7584	0.7345	0.7851