A computing performance assessment of programming languages based on the execution speed and memory usage for matrix diagonalisation

Lorraine Li

5098 Words **Keywords:** Programming Languages, Matrix Diagonalisation
17th August, 2021

Abstract Quick processing of large data sets is needed due to the development of industries include machine learning, medical treatment and aerospace. Matrices are used to represent the transformation of data and points within the data sets, and they become easier to compute with after being written in diagonal matrices by applying matrix diagonalisation. Experiments were carried out to implement matrix diagonalisation with three programming languages, Python, Fortran and Julia. Both time and memory consumption was recorded and used to assess the language performances. The experiments were repeated using different numbers of cores. The time usages of all three languages decreased with the increment of core number, while the trends of memory changes in compiled and interpreted languages were opposite. Based on the comprehensive ability of saving time and memory, Fortran won the computing performance assessment for matrix diagonalisation.

Contents

1 Introduction

1.1 Why performance computing needs to be assessed

Because of the rapid growth of technologies include machine learning (ML) and 3D imaging, there is increasing amount of data, and the size of data sets for analysis is also increasing [1]. When using machine learning for climate prediction, enough previous data needs to be evaluated in order to obtain as accurate result as possible. The more data used, the more reliable the result will be. Therefore, the computing efficiency is required so as to prevent the increase in data volume from slowing down the processing speed.

1.2 Why do people diagonalise a matrix

A matrix consists of numbers arranged in rows and columns, and has a rectangular shape. Each number within it is called an element. A vector can also be seen as a matrix, but with a single row or column. Equation 1 shows the multiplication of a 2×2 matrix and a 2×1 matrix (also a vector), which results in a 2×1 matrix.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} ae + bf \\ ce + df \end{pmatrix} \tag{1}$$

When creating games using 3D technology, the main part is managing the movements of objects. This can be seen as the transformations of points making up the objects [2]. The positions of points can be expressed as vectors, and the process of transformation can be represented by the multiplications of vectors and matrices. However, as data sets get larger, the number of individual calculations required increases greatly. Also, the calculation speed is lowered since there are too many non-zero elements needing to be calculated.

Matrix diagonalisation is the method of writing a matrix into diagonal matrix, which has all elements zero except those on the main diagonal, to minimise the number of non-zero elements. Therefore, the matrix becomes easier to compute with.

1.3 What is a programming language

A programming language is used to produce a set of instructions for the computer to implement a task [3]. The code written in the language is later translated to machine code, which can be understood by the computer, and executed by the processor. The platform of programming is called integrated development environment (IDE), which is a software consisting of comprehensive tools for writing and improving programs. Languages will be assessed by their performances of running programs implementing matrix diagonalisation.

2 Literature review

2.1 Basics of matrix calculation and linear algebra

2.1.1 Matrix multiplication

The shape of a matrix is named after "*number of rows* by *number of columns*", written as "*number of rows* $\times$ *number of columns*". The product of a $m \times n$ matrix and an $n \times p$ matrix will be a $m \times p$ matrix.

Generally, if a matrix **C** is the product of matrix **A** and **B**, then the elements in **C** are calculated according to the Equation 2.

$$\begin{aligned} \mathbf{C}_{ij} &= \mathbf{A}_{i1}\mathbf{B}_{1j} + \mathbf{A}_{i2}\mathbf{B}_{2j} + ... + \mathbf{A}_{in}\mathbf{B}_{nj} \\ &= \sum_{k=1}^{n} \mathbf{A}_{ik}\mathbf{B}_{kj} \end{aligned} \tag{2}$$

2.1.2 Determinant

When an $n \times n$ matrix is applied to a vector, it leads to a transformation, and is written in the form of multiplication. Geometrically, the determinant of a matrix is the factor of area or volume transformed by it. When determinant is equals to zero, it means the transformation leads to a flat plane, a line, or a point.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{3a}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a \\ c \end{pmatrix} \tag{3b}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} b \\ d \end{pmatrix} \tag{3c}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} a+b \\ c+d \end{pmatrix} \tag{3d}$$

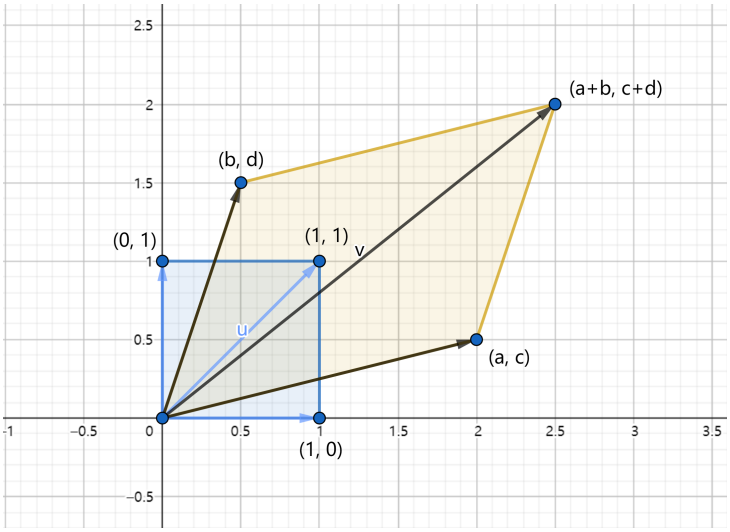

Fig. 1. Transformation of a square by a matrix.

Figure 1 illustrates the transformation shown in Equation 3 geometrically. Beginning with a square formed by vectors $(0, 0)$, $(1, 0)$, $(0, 1)$ and $(1, 1)$, after applying by the matrix, they become $(0, 0)$, (a, c), (b, d) and $((a + b, c + d)$ respectively. The original area is 1 unit area, and after transformation, it becomes the area of the parallelogram formed by the four new vectors, which is $ad - bc$ unit area after simplification. Therefore, the factor of changing the area, is also $ad - bc$.

Equation 4 shows the formula calculating determinant of a 2×2 matrix with different denotations.

$$det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc \tag{4}$$

The process of calculating determinant of a 3×3 matrix is shown in Equation 5.

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \tag{5}$$

2.1.3 Inverse matrix

Geometrically, the inverse of a matrix counteracts the transformation done by the matrix. After multiplying the transformed vector by the inverse matrix of transformation, the vector will return to its original shape.

The determinant of a matrix is used to determine if it has an inverse. The inverse of the matrix only exists when determinant is not zero. Moreover, when multiplying a matrix and its inverse, the result will be an identity matrix. The identity matrix is diagonal with all the elements on its main diagonal equal to one, and everything else is zero. When a matrix is multiplied by the identity matrix, the result is still the original matrix, shown in Equation 6.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{6}$$

The inverse of a 2×2 matrix, $\mathbf{A}$, can be calculated as in Equation 7, denoted by $\mathbf{A}^{-1}$.

$$\mathbf{A}^{-1} = \frac{1}{|\mathbf{A}|} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \tag{7}$$

As for an $n \times n$ matrix, the inverse can be calculated with the following augmented matrix.

$$\left(\begin{array}{ccc|ccc} a & b & c & 1 & 0 & 0 \\ d & e & f & 0 & 1 & 0 \\ g & h & i & 0 & 0 & 1 \end{array} \right) \tag{8}$$

By doing the same operations to the matrices on both sides, when the matrix on the left can be written as an identity matrix, the right-hand side matrix will be the inverse.

2.1.4 The basis vectors

In the Cartesian coordinate system based on the axes x and y, the unit vectors toward the horizontal and vertical axes are the basis vectors. Any of the vectors in two dimension can be expressed by adding the multiples of these two (which is called linear combination). However, it is still possible when the basis vectors are changed.

In Figure 2, vector $\mathbf{x}$ can be expressed using components $\mathbf{x}_1$ and $\mathbf{x}_2$ in the original basis (u_1, u_2). If the basis (v_1, v_2) is used instead, the same vector can still be found, but in an alternative way. After changing the basis, $\mathbf{x}$ is expressed using components $\mathbf{x}'_1$ and $\mathbf{x}'_2$.

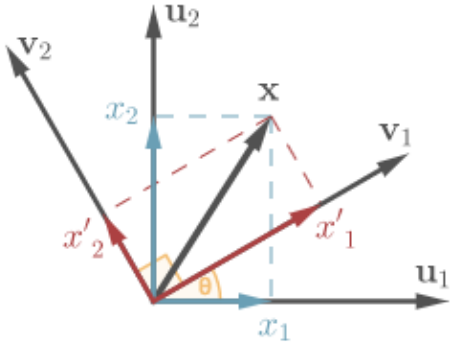

Fig. 2. Basis change example [4].

2.2 Diagonalisation of matrix

2.2.1 Eigenvectors and eigenvalues

A vector will be transformed when multiplied by a matrix. If the process results in unchanged direction, which stretches the original vector by a factor shown in Figure 3, the vector is the eigenvector of the matrix [5]. This is only satisfied with square matrices, where the number of rows and columns are equal. For an $n \times n$ matrix, there will be n eigenvectors.

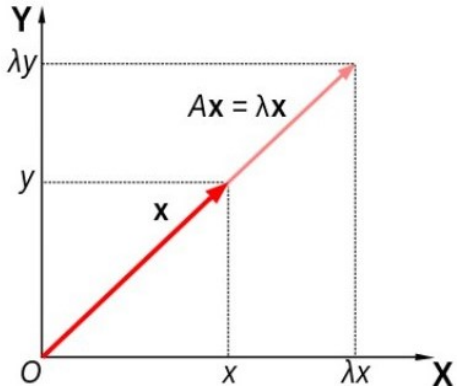

Fig. 3. Example of stretching a vector without changing direction [6].

Each eigenvector has a corresponding eigenvalue, which is the factor of stretching mentioned above. The mathematical relationship is displayed in Equation 9, where **A** still stands for the matrix, **x** stands for the eigenvector and λ is the eigenvalue of **A**.

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{9}$$

2.2.2 Purpose

The purpose of diagonalisation, is to write the matrix of transformation into the form of diagonal matrix [7], which minimises the number of non-zero elements and makes the matrix easier to compute with. This is achieved by transforming the basis of the vector to the one where all the elements that are not on the main diagonal vanish to zero.

Diagonalisation is the process of writing a matrix in the format as Equation 10. **A** stands for the original matrix; columns of **P** are eigenvectors of **A**; and matrix **D** is diagonal, with eigenvalues of **A** on its main diagonal.

The new basis is formed by the eigenvectors in columns of **P**. Illustrating in Equation 11, for a vector representing by the new basis, it is first multiplied by the matrix **P** of change of basis, to get the same vector in the xy coordinates system. Then, after multiplying the matrix **A** of transformation, the transformed vector in the xy coordinates system is returned. Finally, the result is multiplied by the inverse of matrix of the basis change, $\mathbf{P}^{-1}$, to find the transformed vector in the new basis. The representation of **A** in the new basis will therefore be in terms of **D**. Equation 10 is gotten after simplifying and reordering Equation 11.

$$\mathbf{A} = \mathbf{PDP}^{-1} \tag{10}$$

$$\mathbf{P}^{-1}\mathbf{APv} = \mathbf{Dv} \tag{11}$$

Furthermore, with the standard format, calculating powers of matrix **A** will become easier than duplicating matrix multiplication.

$$\begin{aligned} \mathbf{A}^n &= (\mathbf{PDP}^{-1})^n \\ &= \mathbf{PDP}^{-1}\mathbf{PDP}^{-1}...\mathbf{PDP}^{-1} \\ &= \mathbf{PD}^n\mathbf{P}^{-1} \end{aligned} \tag{12}$$

According to the properties that the product of a matrix and its inverse is the identity matrix, and the product of a matrix and an identity matrix is still the original matrix, when breaking the expression down, everything in between cancels out. This will lead to the result in Equation 12, and we only need to take the powers of matrix **D** to get powers of **A**.

2.2.3 Detailed derivation

Starting with Equation 9, Equation 13 can be obtained by multiplying the identity matrix.

$$\begin{aligned} \mathbf{IAx} &= \mathbf{Ax} \\ &= \mathbf{I}\lambda\mathbf{x} \end{aligned} \tag{13}$$

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0} \tag{14}$$

Equation 14 can be further derived after reordering. The equation shows that the product of a matrix and a vector is zero, where the vector **x** is not expected to be zero, and multiple solutions are needed instead. As a result, there will be a many to one mapping to a **0** vector, and the only possible solution is to have a zero determinant, leading to Equation 15.

$$det(\mathbf{A} - \lambda\mathbf{I}) = 0 \tag{15}$$

Since λ is the only unknown item, the expansion of the equation will result in a polynomial of it, which is called characteristic equation. It is solved to find the eigenvalues of matrix **A**, and each eigenvalue is later substituted into Equation 9 to find its corresponding eigenvector, and then written in Equation 10. Equation 16 shows the diagonalisation process of a 3×3 matrix.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} V_{1,1} & V_{2,1} & V_{3,1} \\ V_{1,2} & V_{2,2} & V_{3,2} \\ V_{1,3} & V_{2,3} & V_{3,3} \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} \begin{pmatrix} V_{1,1} & V_{2,1} & V_{3,1} \\ V_{1,2} & V_{2,2} & V_{3,2} \\ V_{1,3} & V_{2,3} & V_{3,3} \end{pmatrix}^{-1} \tag{16}$$

2.3 Computer hardware and software

2.3.1 Hardware

Computer hardware is the "physical part" of a computer which can be touched [8]. It carries out the instructions given by software. Basic hardware components for programming include central processing unit (CPU), random access memory (RAM), and storage.

The CPU is also referred to as processor. It fetches instructions from RAM, reads and processes them and later sends to the related components for execution. Research performed by Ismail [9] has confirmed that plugging extra processors (called cores in this case) is helpful for increasing the computer processing performance. Because when using multiple cores, several processors are working at the same time to implement a task, which therefore reduce the time of implementation.

RAM stores data that is currently in use, and also instructions for the processor to execute [10]. It is directly accessed by the CPU, therefore sometimes it is also called primary memory. Programming requires a lot of memory because of the compiling and debugging processes.

The storage holds data for future use, and it does not exchange data with CPU directly. It is also known as secondary memory since the information needed is always provided by storage and loaded into primary memory.

2.3.2 Software

Software is a collection of instructions telling the computer what to do. There are two common types, application software and system software.

Application software performs functions for specific needs. Platforms that used to write programming codes fall into this category, such as PyCharm and Visual Studio.

System software manages the behaviours of computer hardwares, as for the provision of basic functionalities required by users, or for keeping the application software running well [11]. The most frequently used ones include Microsoft Windows, macOS, Linux.

2.4 Programming

2.4.1 Programming languages

All the programming languages are grouped into two broad categories according to how the programs are implemented, interpreted and compiled.

Interpreted languages need a program called interpreter, which parses the source code (the plain text created using programming languages) and execute it without any transformation. Generally, the interpreter is automatically installed with the programming software.

As for compiled languages, The original code is translated from human-readable format (known as high-level language) to machine-readable format (or machine code) by a program called compiler. It produces an executable file containing the machine code, which will be executed by the processor. Sometimes compilers need to be installed additionally, and different compilers used may affect the efficiency of the program [12].

2.4.2 Typical programming languages

The properties of the following three programming languages are highlighted since they are usually considered to be superior in readability and performance computing. According to the discovery of Jones [13], there are four primary programming languages in data science, which are useful because of their powerful libraries. Within the languages, only one from the pair of languages that can perform similar tasks was selected. Therefore, Python was chosen between Python and R due to its ease of learning. The other two selected were Fortran and Julia. The three programming languages were used to write programs in the subsequent experiment.

2.4.2.1 Python

Python is an interpreted language and famous for its readability. It is appreciated by programmers for its simple syntax and clear structure for a program. Furthermore, the vast amount of modules (a set of codes written for certain tasks that can be reused), packages (a collection of modules) and libraries (a collection of packages) in Python speed up the time of writing a program, especially those with complex logic and operations [14]. The NumPy package meets the majority needs of numerical computations, and it has a separate module for linear algebra, which reduces the time of writing and calculating, however with long-winded function calls [15].

2.4.2.2 Fortran

Fortran is one of the compiled languages, and is preferred by programmers when doing numerical computations due to its efficiency [16]. It first appeared in the 1950s, and has been used for over six decades. Persistent debate exists about whether it will become outdated.

According to the TIOBE index (stands for The Importance of Being Earnest, it is a monthly updated ranking of the top 50 popular programming languages by counting web page hits of the languages on search engines.), it rose sharply since June 2021, and was back into the top 20 [17], stating the incremental demand for the numerical performance of programming languages. Within the large number of numerical libraries written in Fortran, the Linear Algebra Package (LAPACK) is dedicated to solve linear algebra problems.

2.4.2.3 Julia

Julia is also a compiled language, and was invented as a combination of multiple programming languages' features [18], like the straightforward mathematical notations, computational power for linear algebra, and ease of learning [19]. It is relatively new (first appeared in 2012), which results in the lack of libraries [20]. However, it is possible to call libraries and packages from languages including Python and Fortran.

The speed of Julia was proved to be significantly greater than the two mentioned above. When running a coin flip code which outputs the percentage of getting heads, the time usage of Julia is one-tenth as long as Fortran, and one-thousandth of that of Python [21].

2.4.3 Execution speed

In the last century, execution speed was an important factor for performance evaluation of a programming language [22]. Although there is a mass of approaches used to assess the performance of a language, recording execution time of a program is relatively easy to achieve, and it gives researchers the potential benefits on time saving of learning a new programming

language [23].

Compiled languages are usually superior to interpreted languages when comparing only the execution speed (running the compiled codes) [24]. In addition, research showed that the time consumption reduces as the number of cores used increase [25]. To be more specific, in Pisani's experiment [21], four programming languages were used to run a coin flip code with different number of cores used, with the source codes released. All the programs were sped up at different rate as the number of cores used increased. Pisani [21] found that the increase in speed of a Python program slowed down when more than three cores were used; the speed up of a Fortran program was roughly proportional to the increase in the core numbers; whereas time reduction in Julia was not obvious due to the small program size.

With regard to compiled languages, compilers matter as well. For instance, a program written in Fortran works better with Intel Fortran compiler than with GNU Fortran because of less consumption in time [12].

2.4.4 Memory usage

Programming languages have dependencies on memory. It was shown by Singh et al. [26] that some programs that calculate factorials terminated or output wrong answers when computing with big values of upper limits of the factorials. In the experiment, Python was one of the languages that can handle large inputs and maintain its normal computing ability. Also, the run time of the program was limited due to the large consumption of RAM in Fortran in Pisani's experiment [21], which reduced the accuracy of the final results. However, Python and Julia did not have such problem. Therefore, in order to rapidly process the large data sets accurately without unexpected outputs, recording the memory usage is necessary [27]. The memory required to run a program is always allocated from heap, which stands for large amount of available memory and is located in RAM.

According to the memory comparison done by Prechelt [28], the faster the compiled program runs, the more memory it requires. On the contrary, the reversed pattern was found for interpreted languages, programs that cost more memory are prone to be slower.

Libraries will also affect memory usage. They can be divided into two types, static libraries and shared libraries. A static library is copied into the program every time it is being called, whereas a shared library is only copied once and its functions are shared by multiple programs. If they are used only once, the gap in memory usage is not obvious. The more programs using a shared library, the more advantages it has in saving memory [29].

2.5 Conclusion of literature review

The works above illustrate the process of matrix diagonalisation and the benefits it brings. Especially at present, there are huge demands for the efficiency of dealing with large data sets, matrix diagonalisation plays an essential role to help with data processing.

There have been many studies on the computing performance of programming languages and factors that affect their time and memory consumption. Time usage was affected by the number of cores used, and libraries called in the program could change the cost of memory. However, there is still lack of reference for completing laborious computation tasks by using programming languages that are good at numerical calculations and cherished by programmers.

Therefore, an experimental plan was formulated to help compare the computing performance of languages that are powerful in data processing.

In the experiment, the computing performance of Python, Fortran and Julia will be assessed by recording and comparing the execution time and memory usage of programs implementing matrix diagonalisation. The three programming languages are used due to their ease of learning, as well as the ability of data processing generated by the libraries within them. The modules used for timing the programs in the work of Pisani [21] will be referenced. Furthermore, by using different number of cores for execution, whether the difference in language performances will be changed can also be observed.

3 Experimental method

3.1 Procedures and purpose

A 3×3 matrix was populated by using the random library in Python to randomly select the elements within it, each of them is between 0 and 10. The matrix shown below is the first matrix that has three different eigenvalues. It was recorded and used as inputs in the programs.

$$\begin{pmatrix} 1 & 4 & 7 \\ 5 & 9 & 4 \\ 3 & 3 & 1 \end{pmatrix} \tag{17}$$

Programs were written in three programming languages to diagonalise the same matrix. According to the process of diagonalisation illustrated in section 2.2, the eigenvectors were calculated by functions and defined as a new variable. The variable was later calculated by functions of inverse matrix and matrix multiplication to get a diagonal matrix using different basis. The results were combined to write the original matrix in a new form for easier calculations.

Then, their execution time and memory usages were recorded and compared. The number of cores used was changed from 1 to 8, and the operations were repeated to get eight groups of data. Finally, the data was presented in a table, and the comparisons were summarised, as well as the change in performance as switching the core numbers.

3.2 Programming

3.2.1 Selection of programming languages

As mentioned in section 2.4.2, Python is popular because of its readability and abundant libraries; Fortran is welcomed when doing numerical calculations, and it also has vast number of libraries; Julia has the combination of features of multiple languages, and it is outstanding through comparison of execution speed. Therefore, Python, Fortran and Julia were used in the experiment. The NumPy package in Python, LAPACK in Fortran and LinearAlgebra library in Julia were called to write programs of matrix diagonalisation.

3.2.2 Libraries

All the libraries used in Python are shared libraries, since the modules and functions within them are referenced using the import statement. The functions within the modules are called by full name references, which therefore lead to lengthy function calls. In Fortran and Julia programs, the libraries used are all static, since the codes are directly copied into the programs and compiled with them.

3.2.3 Functions

In Python, the np.linalg.eig function was called to get the eigenvectors, the np.linalg.inv and the np.dot functions were used to make calculations with the matrix. In Fortran, the eigenvectors were calculated by calling the geev function, the functions of getrf and getri were used to calculate the inverse matrix, and matrix multiplications were done by using the matmul function. In Julia, the eigvecs and inv functions were called to get the eigenvectors and inverse matrix, and the matrix multiplications were achieved by joining the two matrices with a multiplication sign.

3.3 Experimental environment

3.3.1 Computer system

Table 1
Hardware and software of the computer system.

Hardware and Software	Detailed Information
CPU	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz
RAM	8.00 GB (7.78 GB available)
System software	Windows

The hardware and software of the computer system is shown in Table 1, and there are 8 available cores in total. The number of cores used was changed through the System Configuration tool, which was opened by searching "msconfig" from the taskbar. Under the Advanced Boot Option, the number of processors was chosen.

3.3.2 Integrated development environment(IDE)

The IDE used for Python was PyCharm, and its self-contained interpreter was used for the program. The Fortran program was run in Microsoft Visual Studio, and was compiled by an extra installed compiler called Intel Fortran. The code written in Julia used the Juno IDE built in a code editor (for writing and editing the code, but without any functionality) called Atom, and was compiled by the Juno compiler.

3.4 Recording techniques

3.4.1 Timing standard

The modules used in Pisani's experiment [21] were used to measure the time consumption. In Python, the time module was called and used for recording time. The cumulative usage of time was taken at the beginning and at the end of the code. The net execution time was calculated by subtracting the two values.

The OpenMP (stands for Open Multi-Processing) library was used for time measurement in Fortran. It is similar to the time module in Python. The time was recorded twice, one at the beginning and one at the end of each run, and the execution time was the difference between the two.

The @time macro (usage of short statements instead of frequently used code lines) was used in the Julia code. It can only measure the time usage of a function, so the code was required

to be written in a single function to ease the measurement.

The Fortran and Julia programs were compiled in advance, because only execution time was required for the comparisons. In order to obtain accurate and stable data as much as possible, each program was run 5 times as a warm-up. Then, average value of 10 runs was recorded.

3.4.2 Memory measurement

For the Python program, an additional module called psutil (stands for process and system utilities) was downloaded and used. The total memory usage was output in MiB (mebibyte, a unit representing the number of bytes with base of 2). The memory profiler module was also used for double-checking. The @profile macro was written directly above the function that performs matrix diagonalisation, and the cumulative memory at each row of the function was output as a table.

As for Fortran, according to the Microsoft documentation for measuring memory in Visual Studio, memory can be viewed in heap profiling. Breakpoints were set at the beginning and at the end of the program, and the debugging mode was used. The window of the diagnostic tools was opened, and heap profiling was chosen. Under the toolbar option of memory usage, snapshots were taken at the positions of the breakpoints, and the change of heap size used was displayed in KB (kilobyte). This was recorded as the memory usage of the program after being converted into MiB.

In Julia program, the Profile library was called for memory measurement due to the record on the official website of Julia. The @profile macro was used, and its way of application is similar to that of Python. It was placed above the function for the diagonalisation process, however, only the total memory used was output in KiB (kibibyte, also a unit with base of 2). The results were later converted into MiB to maintain the consistency of the units.

3.4.3 Determination of the accuracy of results

All the units of data recorded as execution time were turn into ms (millisecond), and those of memory usage were in MiB. Based on the standard of International System of Units (SI) prefixes [30] and Institute of Electrical and Electronics Engineers (IEEE) 1541 [31], KB has the base of 10 while KiB and MiB are bytes of powers of 2. Therefore, KB was converted to MiB by timing 10^3 and then dividing by 2^{20}, KiB was converted by dividing by 2^{10}.

As few decimal places as possible were used to avoid data equality or insignificant change after rounding. The results of time were in 4 sf (significant figure), and data of memory was in 5 sf.

The calculation of matrix diagonalisation was also done manually, in order to further ensure correct results were output by the programs, which was also helpful for coding the intermediate steps of the programs.

4 Results and discussion

4.1 Experimental data

Table 2
Data collected for three programming languages using different number of cores.

Programming Languages	IDE	Compiler/ Interpreter	Core Number	Execution Time (ms)	Memory Usage (MiB)
Python	PyCharm	PyCharm	1	427.6	24.968
			2	424.2	24.952
			3	377.5	25.173
			4	376.2	25.101
			5	375.2	25.073
			6	373.7	25.067
			7	373.3	25.063
			8	372.8	25.045
Fortran	Microsoft Visual Studio	Intel Fortran	1	16.66	0.13780
			2	13.38	0.13818
			3	11.38	0.13958
			4	11.05	0.14173
			5	10.58	0.14244
			6	10.43	0.14274
			7	10.48	0.14647
			8	10.02	0.15078
Julia	Atom	Juno	1	4.795	0.90126
			2	4.743	0.90129
			3	4.671	0.90138
			4	4.626	0.90140
			5	4.545	0.90143
			6	4.461	0.90147
			7	4.440	0.90158
			8	3.471	0.90181

4.2 Analysis of experimental data

Line graphs of time and memory changes in three programming languages were drawn according to Table 2. Overall, when looking at the lines representing time in Figure 4-6, Python takes about forty times as long as Fortran, which is more than three times as long as Julia. Although the gaps between their performances are not as exaggerated as the results in Pisani's experiment [21], the two compiled languages are dominant in the comparison of time, and Julia wins among the two. The speed up rate of the Python program slows down to less than one percent after three or more cores are used, which matches the results in the experiment above. The time change of Julia program is not obvious, which is also in line with the results figured out by Pisani. However, the trend of time changing in Fortran is not linear with the increment of core number. Instead, the speed up changes to fluctuation within two percent when more than three cores are used.

The memory changes in three programming languages are also shown in the graphs. According to Figure 4, the change in memory usage of Python program is in a fluctuating state at first. From the use of three cores, the memory usage decreases continuously at a slowing rate. In Figure 5 and 6, both Fortran and Julia programs have a steady upward trend

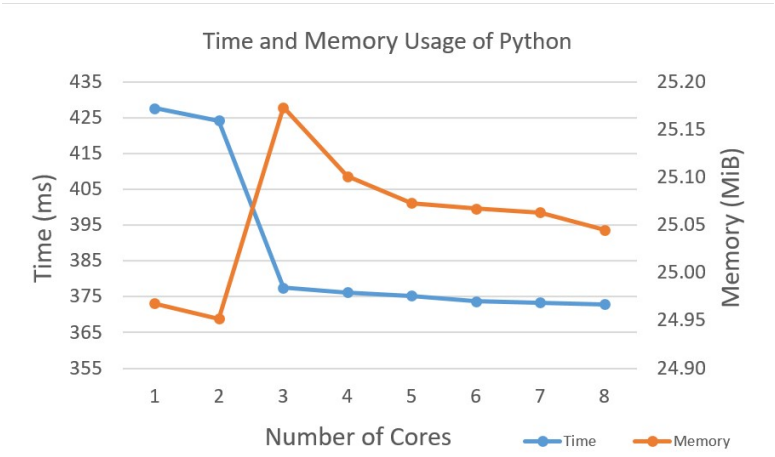

Fig. 4. When more than three cores are used, the speed up rate of Python program slows down, and the state of change in memory usage changes from fluctuation to a downward trend. Both time and memory have a dramatic change when the core numbers are changed from 2 to 3.

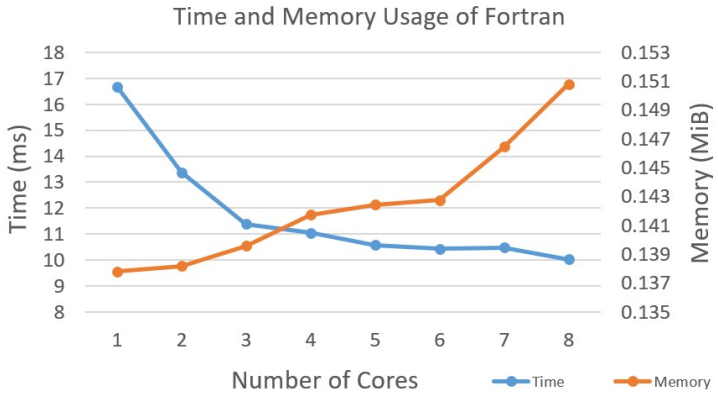

Fig. 5. The speed up in Fortran program changes to fluctuation when more than three cores are used, while the memory change shows a steady upward trend. The time and memory changes have general trends in opposite directions.

throughout the experiment, and Julia has a smaller variation than Fortran. Taken as a whole, the memory usage of Python mostly decreases with the increase of core number, and the other two have a gradually increasing memory usage. Since all the three languages have a downward trend for time consumption, the result is consistent with Prechelt's conclusion [28]. Furthermore, because only one program was written and run in each programming language, it is hard to judge the impact of using different types of libraries on memory usage. Therefore, the data deviation and unfairness of the comparison caused by the types of libraries were ignored.

When comparing the use of time and memory at the same time, Python fails to take advantage of both, and Julia and Fortran win in saving time and memory respectively. Although the Fortran program is more than twice as slow as Julia, its memory usage is about one seventh of that of Julia. Therefore, based on the comprehensive ability of saving time and memory, Fortran wins in the computing performance assessment for matrix diagonalisation.

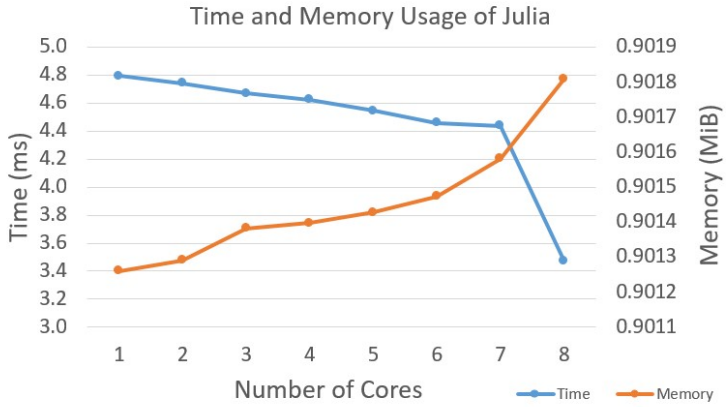

Fig. 6. The time usage of Julia program has an unobvious slow decline, and the memory usage increases gradually. The time and memory have opposite trends of change, and they both have a great change when the core numbers are changed from 7 to 8.

4.3 Limitations

In order not to make the experiment too complex and difficult to manage, only one matrix and three programming languages were used to carry out matrix diagonalisation. This could lead to a lack of accuracy of the data because it was not proved whether the same conclusion is applicable to other matrices. Moreover, there were only two compiled languages and one interpreted language, which led to the limitations of the conclusion comparing the two types of programming languages.

References for algorithms of memory measurement in the programming languages were not found. Therefore there was no reference when writing the programs, and only the trend of the final data can be used to judge whether the results are reasonable. Although heap profiling is recommended on the official website, and was used to measure the memory used in Fortran program, all the memory needed are allocated from the heap. Even if the difference in heap size used was recorded, it might still be more than just the part consumed by the program.

The standards used for the units of memory usages in the three programming languages were not considered. Conversion were done based on the units of raw data output in each language. As a result, the accuracy of the memory data may be threatened because different base numbers were used for the byte numbers with binary and decimal prefixes.

5 Conclusion

In current technologies such as machine learning, it is necessary to analyse large data sets to provide convincing predictions include user preferences and weather forecast. Matrix diagonalisation was introduced to help quickly computing with the data sets. The process of matrix diagonalisation was illustrated in the essay, as well as the execution of a program.

Factors affecting the time and memory consumption of a program were concluded from the literature. The time usage of a program was found to be affected by the number of cores used, and the types of libraries used might influence the memory usage. Based on the discoveries, an experiment was carried out for the performance assessment of Python, Fortran and Julia

comparing the time and memory usage, with different number of cores used. Finally, Fortran beat out the other two languages because of its advantage of saving both time and memory, and Julia could be seen as a strong competitor. Although Fortran was more than twice as slow as Julia, its memory usage was about one seventh of that of Julia. And Python took forty times as long as Fortran, and occupied hundreds of times more memory than Fortran did.

In the future, Fortran is likely to be replaced by languages that can compute faster, such as Julia. However, it is unrealistic in the short term, since Julia is not mature enough as a young language with insufficient libraries. After being upgraded, Julia is expected to be more popular when doing numerical calculations due to its ability of saving execution time. However, Fortran is still outstanding in terms of the ability of saving both time and memory so far. Python will continue to be welcomed by both professionals and non-professionals in computer science, because it is easy to learn and there are adequate supplies of libraries to help programming.

Although the limitations discussed may affect the accuracy of the data, the conclusions of the research question may be used as a reference for future studies. The performance of a programming language will be expected to be even more important in the future. Therefore, further research about the performance of programming languages is needed due to the quick upgrading, such as the new libraries developed and newly invented languages.

6 References

[1] V. Chawla, *Why High Performance Computing Could Become The Next Frontier For Enterprise AI*, Sep. 2020.

[2] T. Yip, *Matrices in Computer Graphics*, Mar. 2001.

[3] A. A. Aaby, "Introduction to Programming Languages," p. 233, Jul. 2004.

[4] B. Belousov, *Change of basis vs linear transformation*, May 2016.

[5] L. Mckelvey and M. Crozier, *Cambridge International AS & A Level Further Mathematics Coursebook*. Cambridge: Cambridge University Press, Nov. 2018.

[6] Wikipedia, *Eigenvalues and eigenvectors*, Page Version ID: 1031302158, Jun. 2021.

[7] M. Taboga, *Matrix diagonalization*, 2017.

[8] G. W. Cage, "Computer Hardware," *Dermatologic Clinics*, Computers in Dermatology, vol. 4, no. 4, pp. 533–543, Oct. 1986.

[9] D. Ismail, "Multicore Processor Performance Analysis - A Survey," p. 9, Apr. 2013.

[10] C. Petzold, *Code: the hidden language of computer hardware and software*. Redmond, Wash: Microsoft Press, 2000.

[11] S. Langfield and D. Duddell, *Cambridge International AS and A level Computer Science coursebook*. Cambrige, United Kingdom: Cambridge University Press, 2015.

[12] L. E. Young-S., P. Muruganandam, S. K. Adhikari, V. Lončar, D. Vudragović, and A. Balaž, "OpenMP GNU and Intel Fortran programs for solving the time-dependent Gross–Pitaevskii equation," *Computer Physics Communications*, vol. 220, pp. 503–506, Nov. 2017.

[13] M. T. Jones, *The languages of data science*, Apr. 2018.

[14] T. E. Oliphant, "Python for Scientific Computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[15] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, Mar. 2011, Conference Name: Computing in Science Engineering.

[16] J. Moreira and S. Midkiff, "Fortran 90 in CSE: A case study," 1998.

[17] *Index — TIOBE - The Software Quality Company*, Jul. 2021.

[18] E. Jeff Bezanson, *Why We Created Julia*, Feb. 2012.

[19] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A Fast Dynamic Language for Technical Computing," *arXiv:1209.5145 [cs]*, Sep. 2012, arXiv: 1209.5145.

[20] J. Danielsson and J. R. Fan, *Which numerical computing language is best: Julia, MATLAB, Python or R? — VOX, CEPR Policy Portal*, Jul. 2018.

[21] W. Pisani, *Comparing Julia, C++, Fortran, and Python Run Times with Coin Flip Code*, Jul. 2018.

[22] 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*, Vancouver BC Canada: ACM, Oct. 2017, pp. 256–267.

[23] S. B. Aruoba and J. Fernández-Villaverde, "A Comparison of Programming Languages in Economics," National Bureau of Economic Research, Cambridge, MA, Tech. Rep. w20263, Jun. 2014, w20263.

[24] A. Krauss, *Programming Concepts: Compiled and Interpreted Languages*, Jul. 2015.

[25] D. Datta, D. Mittal, N. P. Mathew, and J. Sairabanu, "Comparison of Performance of Parallel Computation of CPU Cores on CNN model," in *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, Vellore, India: IEEE, Feb. 2020, pp. 1–8.

[26] P. Singh, S. Shukla, S. Chandra, and V. Dixit, "Performance evaluation of programming languages," in *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, Coimbatore: IEEE, Mar. 2017, pp. 1–4.

[27] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics," *BMC bioinformatics*, vol. 9, p. 82, Feb. 2008.

[28] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, Oct. 2000.

[29] D. Huang, "An analysis of how static and shared libraries affect memory usage on an IP-STB," p. 45, Sep. 2010.

[30] NIST, *Metric (SI) Prefixes*, text, Last Modified: 2021-06-24T09:39-04:00, Jan. 2010.

[31] IEEE, "IEEE Standard for Prefixes for Binary Multiples," IEEE, Tech. Rep., Dec. 2002, ISBN: 9780738161068 9780738161075.