QUANTUM INFORMATION AND COMPUTING

Assignment 3

Author
Marco Chiloiro
November 21, 2023

1 Scaling of the matrix-matrix multiplication

This section focuses on the scaling analysis of matrix-matrix multiplication. The primary objectives are to explore how the execution time of the multiplication operation scales with the input size N and to compare different multiplication methods.

1.1 Theoretical background

Fortran stores matrices in column-major order, also known as column-wise order. In this storage format, elements of a matrix are stored column by column in memory.

Given two square matrices of size N, A and B, the standard way to compute the matrix-matrix multiplication between them is

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j}, \tag{1}$$

where C is the resulting matrix. From now on, this method will be call *row by column* method.
An equivalent method is

$$C_{i,j} = \sum_{k=1}^{N} A^T_{k,i} B_{k,j}, \tag{2}$$

the *column by column* matrix-matrix multiplication method.
By implementing these methods in Fortran, the number of elementary operations for both methods is on the order of N^3, but due to Fortran's storage method it is reasonable to expect that the second algorithm will be more efficient than the first one.

1.2 Implementation

The aforementioned methods, along with the built-in Fortran `MATMUL` function, have been implemented within a Fortran function contained in a `.f90` file that gives in output the execution time of the matrix-matrix multiplication, given the size N of the square matrices and the selected method. The compilation of this file is performed using `f2py3`, a tool which is part of the `NumPy` project designed for wrapping Fortran code into Python. For this compilation, the default `-O1` optimization flag has been used.
A Python script changes N between the two values $N_{min} = 100$ and $N_{max} = 5000$, launches the previously compiled program and stores the results of the execution time in different files depending on the multiplication method used. Another Python script imports the data, fits the scaling of the execution time for different methods as a function of the input size, and plots the results.
By assuming the relationship $t = c \cdot N^m$, where t is the execution time and c, m are parameters, the data are fitted by performing a linear fit on the logarithmic transformation of the equation, i.e., $\log_{10}(t) = \log_{10}(c) + m \cdot \log_{10}(N)$.

1.3 Results

In Figures 1, 2, and 3, the observed trends align with expectations. The most efficient matrix-matrix multiplication algorithm appears to be the built-in Fortran `MATMUL`, exhibiting execution time scaling proportional to $O(N^{2.83})$, followed by the column-by-column method with $O(N^{3.04})$ scalability, while the row-by-row approach exhibits the worst scalability of $O(N^{3.38})$.

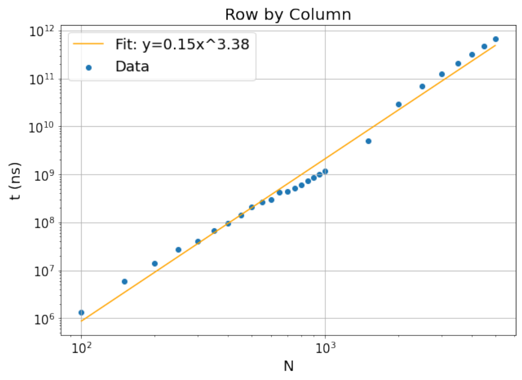

Figure 1: Log-log scale plot of the row-by-column method execution times and the relative linear fit.

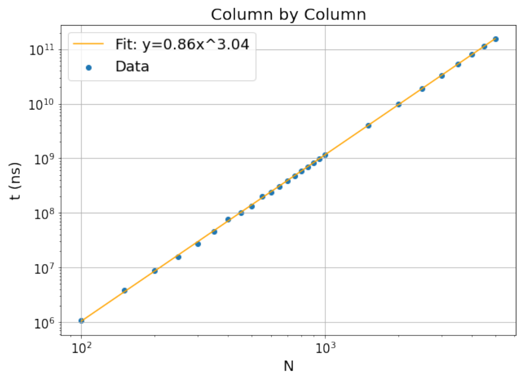

Figure 2: Log-log scale plot of the column-by-column method execution times and the relative linear fit.

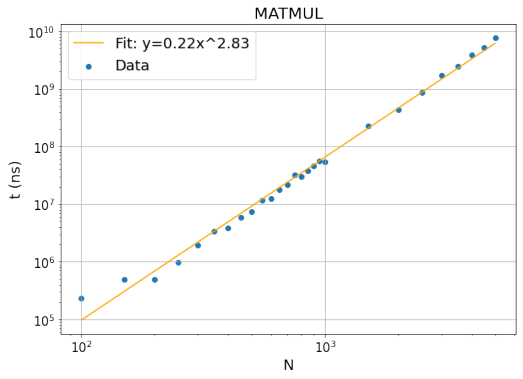

Figure 3: Log-log scale plot of the `MATMUL` method execution times and the relative linear fit.

2 Random matrix theory

This section focuses on the probability distribution of normalized spacing between the eigenvalues of random Hermitian matrices. Additionally, it investigates how this distribution contrasts with the normalized spacing between eigenvalues in a diagonal matrix that possesses random real entries.

2.1 Theoretical background

The normalized spacing s_i between the eigenvalues λ_i (sorted in ascendig order) of a Hermitian matrix are defined as:

$$s_i = \frac{\Lambda_i}{\Lambda_{avg}}, \tag{3}$$

where $\Lambda_i = \lambda_{i+1} - \lambda_i$ and Λ_{avg} is the average of Λ_i. From random matrix theory, we expect that the normalized spacing s of a random Hermitian matrix, whose entries are drawn from a normal distribution, adheres to a distribution $P(s)$ characterized by the form

$$P(s) = as^{\alpha} e^{bs^{\beta}}, \tag{4}$$

where a, b, α, β are parameters. From the *Wigner surmise* statement, we expect that $a > 0$, $\alpha \simeq 2$, $b < 0$ and $\beta \simeq 2$.
In the case of a diagonal matrix that possesses random real entries, since these are drawn from a normal distribution, we expect that the normalized spacing follow an exponential distribution, i.e.

$$P(s) = ae^{bs}. \tag{5}$$

2.2 Implementation

For this task, we employ a Python script that iteratively generates ($n_{rep} = 20$ times) Hermitian random matrices of size $N = 1000$. The matrix entries are sampled from a normal distribution $N(1, 1)$ (by using `numpy.random.uniform` function). For each iteration, the script computes the eigenvalues (`numpy.linalg.eigvals`) of the generated matrix, sorts them, and calculates the normalized spacing s_i. Subsequently, it constructs a histogram from the collected data to estimate the distribution $P(s)$. The final step is to fit a parametric model as described in the previous subsection (`scipy.optimize.curve_fit`). A similar procedure is performed for the case of real diagonal matrices.

2.3 Results

Figure 4 and Figure 5 visually demonstrate the robust performance of the fitting process on the data. Notably, the first fit yields values for $\alpha = 2.56$ and $\beta = 1.34$, which are both near 2 as expected.

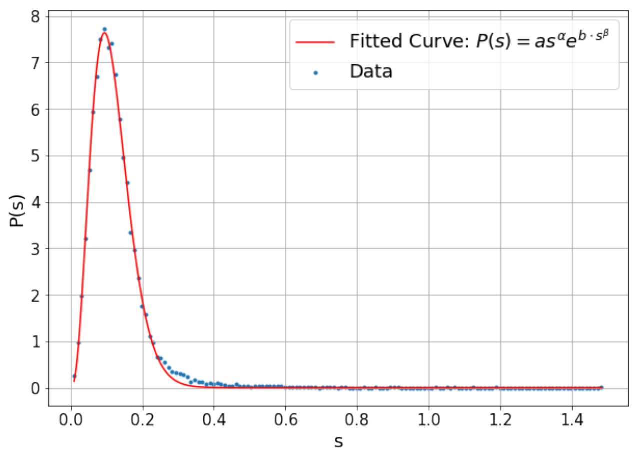

Figure 4: Probability distribution $P(s)$ of the normalized spacing between the eigenvalues of random Hermitian matrices of size $N = 1000$. $P(s)$ was estimated by considering $n_{rep} = 20$ repetitions. The obtained parameters are: $a = 2.28 \cdot 10^4$, $b = -4.56 \cdot 10$, $\alpha = 2.56$ and $\beta = 1.34$.

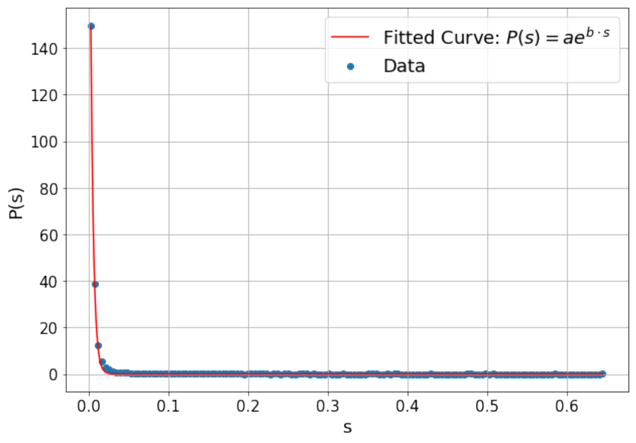

Figure 5: Probability distribution $P(s)$ estimation of the normalized spacing between the eigenvalues in a diagonal matrix that possesses random real entries of size $N = 1000$. $P(s)$ was estimated by considering $n_{rep} = 20$ repetitions. In red the parametric fit. The obtained parameters are: $a = 28.74 \cdot 10$, $b = -28.47 \cdot 10$.