Assignment 3 - Python scripting, random matrices and eigenproblem.

Calandra Buonaura Lorenzo
(Dated: November 11, 2024)

This assignment explores the computational scaling and statistical properties of matrix operations and random matrices. It first investigates matrix-matrix multiplication in Fortran, highlighting different implementation strategies and their efficiency in terms of performance scaling with matrix size N. The work emphasizes the importance of optimizing these operations in scientific computing and provides insights into their computational complexity. Additionally, Python scripting is employed to automate the execution of experiments, manage parameters, and streamline the collection of performance data. The second part of the assignment delves into random matrix theory, specifically examining the properties of eigenvalues in random Hermitian and diagonal matrices. The statistical behavior of eigenvalue spacings, particularly their distribution and repulsion, is analyzed through numerical simulations and compared with the Wigner-Dyson distribution.

1. INTRODUCTION

Matrix-matrix multiplication is a cornerstone operation in various fields, including scientific computing, physics, and engineering. The complexity of this operation scales with the matrix size ($O(N^3)$), and the efficiency of different multiplication implementations is crucial for performance optimization.

In this assignment, we explore the theoretical and computational aspects of matrix multiplication, focusing on its scaling behavior and practical implementation in Fortran90: in particular we compare the performances and the scaling of intrinsic functions and algorithms implemented by-hand. We then transition to the use of Python scripting to facilitate experimentation with different matrix sizes, enabling us to systematically study the computational performance of matrix operations. This analysis highlights the need of a good intrinsic optimization, which is able to leverage other optimized libraries (like BLAS), in order to obtain more computational power.

In addition to matrix multiplication, this assignment introduces random matrix theory (RMT), which investigates the statistical properties of matrices with random entries. Specifically, we examine the eigenvalues of random Hermitian and diagonal matrices, analyzing their spacing distributions: Hermitian matrices exhibit eigenvalue repulsion, which is a distinctive feature of their statistical properties, while diagonal matrices do not. Using numerical simulations, we compare the observed eigenvalue spacing distributions with theoretical models, such as the Wigner-Dyson distribution, to evaluate the applicability of random matrix theory in this context.

2. THEORETICAL FRAMEWORK

This assignment covers multiple topics: first, it examines matrix-matrix multiplication and its computational scaling with matrix size N, focusing on the efficiency and performance implications of different implementations.

Next, Python scripting is introduced to automate tasks and manage experimental parameters, including scaling studies of matrix-matrix multiplication.

Finally, random matrix theory and its associated eigenproblems are explored; the assignment investigates the statistical properties of eigenvalues, in particular related to spacing distributions of eigenvalues.

A. Matrix-Matrix Multiplication

Matrix-matrix multiplication is a fundamental operation in linear algebra, commonly used in scientific computing, physics, engineering, and computer science. Given two matrices A and B of dimensions $N \times M$ and $M \times P$, respectively, their product $C = A \cdot B$ results in an $N \times P$ matrix C. Each element c_{ij} of C is calculated as follows:

$$c_{ij} = \sum_{k=1}^{M} a_{ik} \cdot b_{kj} \tag{2.1}$$

where a_{ik} and b_{kj} are elements of matrices A and B, respectively, and the summation is performed over the index k [1].

Matrix Multiplication in Fortran

In Fortran, matrix-matrix multiplication can be implemented using the intrinsic `matmul` function, which is optimized for high performance and is convenient for large-scale computations. The usage of `matmul` is straightforward:

```
C = matmul(A, B)
```

For finer control over the computation, one can also use an explicit loop-based approach in Fortran, where each element c_{ij} is computed by iterating over the matrix dimensions. An example of this approach, following the classical row by column rule, is shown below:

```
do i = 1, N
  do j = 1, P
    do k = 1, M
      C(i, j) = C(i, j) + A(i, k) * B(k, j)
    end do
  end do
end do
```

This loop-based approach is particularly useful for more complex applications, where additional operations might be required during the multiplication process or for cases that require careful memory management. Nevertheless, this implementation is not as optimized as the intrinsic `matmul` function, which makes use of underlying hardware optimizations and can leverage optimized mathematical libraries, such as BLAS (Basic Linear Algebra Subprograms). These libraries are specifically designed to handle large-scale matrix operations efficiently and can take advantage of CPU cache, vectorized operations, and parallel computing capabilities [2].

We must also highlight that, in Fortran, arrays are stored in column-major order, meaning elements in the same column are stored in contiguous memory locations; thus, this storage format makes a column-oriented approach, like the `kij` loop ordering, more cache-efficient. By iterating over columns in the innermost loop, the `kij` ordering aligns with Fortran's memory layout, potentially leading to better performance compared to row-major loop orderings.

Scaling with Matrix Size N

The computational complexity of matrix-matrix multiplication grows with the size of the matrices. For square matrices of dimension $N \times N$, each element of the resulting matrix C requires N multiplications and $N-1$ additions, resulting in a time complexity of $O(N^3)$ [3]. Consequently, the computational time increases rapidly, making matrix multiplication a computationally intensive operation (even if the scaling is only polynomial). To empirically determine the scaling behavior of matrix-matrix multiplication, execution times can be measured for various matrix sizes N and fitted to a model function. Given the theoretical expectation of cubic complexity, we assume that the relationship follows a power law ($T(N) = kN^3$), where $T(N)$ represents the execution time, N is the matrix dimension, and k is a proportionality constant. To facilitate fitting, we take the logarithm of both sides, yielding a linear equation:

$$\log(T(N)) = \log(k) + 3\log(N) \tag{2.2}$$

This transformation allows us to perform a linear fit to the data when plotted on a log-log scale. In this log-log plot, the slope of the line should correspond to the exponent of N (in this case, around 3), indicating the scaling of execution time with N.

B. Python scripting

Python is a versatile, high-level programming language widely used in scientific computing, data analysis, and engineering applications due to its simplicity and extensive library ecosystem. In particular, python scripts are particularly valuable for tasks that require running programs with varying input parameters, such as in our matrix-matrix multiplication scaling study. By writing a script that varies the matrix size N and collects execution times, we can streamline data collection for different cases, eliminating the need for manual intervention.

This can be achieved using the Python `subprocess` module, which allows us to compile Fortran code, execute programs, and capture output without leaving the Python environment [4]. For example, to compile a Fortran file, we can use the following code:

```
import subprocess
subprocess.run(
    ["gfortran", "source_file.f90", "-o",
    "executable"], check=True)
```

This command compiles `source_file.f90` and creates an executable named `executable`. The `check=True` argument ensures that any errors during compilation are immediately reported, allowing us to catch and handle issues in real time. Similarly, we can run the compiled executable with specific input parameters and capture the output. For instance:

```
result = subprocess.run(
        ["./executable"], input="100",
        text=True, capture_output=True
        )
print(result.stdout)
```

This approach enables us to modify input parameters on the fly, such as varying the matrix size N, and to record the corresponding outputs, which can then be analyzed or saved for further processing.

C. Random Matrices

Random matrix theory (RMT) is a field of mathematics and physics that studies the statistical properties of matrices with random entries. In this assignment, we focus on the statistical behavior of eigenvalues of random matrices, particularly in the context of eigenvalue spacing distributions; we study two types of matrices:

- **Random Hermitian matrix:** A random Hermitian matrix is a complex square matrix H that satisfies $H = H^\dagger$, where $H^\dagger$ is the conjugate transpose of H. In such matrices, the entries are generally complex numbers, with the diagonal elements being real and the off-diagonal elements following symmetry properties of the Hermitian form. The eigenvalues of a Hermitian matrix are real, which allows for the study of their spacing distributions: in particular, Hermitian matrices exhibit eigenvalue repulsion, which means that the expected distribution is not peaked around 0 but around an higher value.

- **Random diagonal matrix:** A random diagonal matrix is a square matrix where all non-diagonal elements are zero, and the diagonal elements are random values. Diagonal matrices are useful in modeling systems where each element acts independently, as the eigenvalues of a random diagonal matrix correspond directly to the random entries along the diagonal; thus, unlike random Hermitian matrices, random diagonal matrices do not exhibit eigenvalue repulsion since no interaction between eigenvalues exists.

One key metric of interest in random matrix theory is the eigenvalue spacing distribution, which quantifies the statistical distribution of spacings s between consecutive eigenvalues [5]. The normalized spacing are defined as:

$$s_i = \frac{\Lambda_i}{\bar{\Lambda}} \quad \text{with} \quad i = 0, n-1 \tag{2.3}$$

where $\Lambda_i = \lambda_{i+i} - \lambda_i$ is the spacing between consecutive eigenvalues and $\bar{\Lambda} = \frac{1}{n-1}\sum_i \Lambda_i$ is the average spacing.

For random matrices, the eigenvalue spacings tend to follow the Wigner-Dyson distribution:

$$P(s) = a\, s^{\alpha} \left(b\, e^{\beta}\right) \tag{2.4}$$

where $P(s)$ represents the probability density function of the spacing s between consecutive eigenvalues. This distribution is able to capture the phenomenon of eigenvalue repulsion, where small spacings are rare, reflecting the underlying symmetries of the hermitian random matrix. In this assignment, eigenvalue spacings are computed by first finding the eigenvalues of each random matrix, sorting them in ascending order, and calculating the differences between consecutive eigenvalues. To analyze the spacing distribution, we construct a histogram of these spacings and compare it with the Wigner distribution given by Equation (2.4); this can provide insights into whether the eigenvalues of the generated random matrices exhibit the expected statistical behavior.

3. CODE DEVELOPMENT

A. Matrix multiplication

For the matrix multiplication, first of all the Fortran module `matrix_matmul.f90` has been developed; it implements matrix multiplication using three different methods, *row-by-column*, *column-by-row*, and the intrinsic *matmul* function. The module contains the following main subroutines:

- `rowbycolumn(A, B, C)`: Multiplies two matrices A and B by iterating over rows of A and columns of B and stores the result in C.
- `columnbyrow(A, B, C)`: Multiplies matrices A and B by iterating over columns of A and rows of B, storing the result in C.
- `timing(n)`: Measures the execution time of the three multiplication methods for a square matrix of size n. It initializes matrices A and B, performs the multiplications, and outputs the elapsed time for each method: "RC" for row-by-column, "CR" for column-by-row, and "I" for the intrinsic `matmul`.

The subroutine `timing` also checks for correct matrix dimensions, memory allocation, and proper deallocation of matrices. It ensures the correct execution of each multiplication method, providing reliable results. The output includes the time taken for each method, which can be used to analyze the performance and scaling behavior with different matrix sizes.

This module is then called from another Fortran file which takes the value of n (the size of the matrix is $n \times n$) as input. This program is compiled and executed thanks to a python script, `python_script.py`, which tests the scaling of matrix-matrix multiplication performance with varying matrix sizes. The script requires three command-line arguments:

- `Nmin`: Minimum matrix size.
- `Nmax`: Maximum matrix size.
- `m`: Number of matrix sizes to test between `Nmin` and `Nmax`.

The script then loops through `m` values between `Nmin` and `Nmax`, running the Fortran program for each value of `N`, and saves execution times for each method in separate text files. We can run the script with the following command:

```
python script.py Nmin Nmax m
```

The function present in the script are:

- `compile_module`: Compiles a Fortran module.
- `compile_fortran`: Compiles the main Fortran program; the optimization flag is fixed at `-O3`, which provides the best possible optimization.
- `run_executable`: Runs the Fortran executable and captures output
- `save_to_file`: Saves the execution time data to a file.
- `main`: Coordinates the execution and data storage.

All the obtained data are then analyzed using a Jupyter Notebook, where the main plot is visualized.

Moreover, the `debugger` module provides a subroutine for debugging purposes; it includes a `checkpoint` subroutine that prints a custom message or a default one when a checkpoint is reached in the program, provided that debugging is enabled.

B. Random matrices

For the random matrix theory analysis, a python script `ranmat` containing the functions has been developed:

- `random_herm_spectrum`: Generates a random Hermitian matrix of size $N \times N$, computes its eigenvalues and eigenvectors, and optionally prints them.
- `random_diag_spectrum`: Generates a random diagonal matrix, computes the eigenvalues and eigenvectors, and optionally prints them.
- `compute_norm_spacing`: Computes normalized spacings between consecutive eigenvalues for either Hermitian or diagonal matrices, with an option to remove the first eigenvalue.
- `compute_spacing_distr`: Computes the eigenvalue spacing distribution across multiple matrices, using the specified matrix type.
- `wigner_dyson`: Defines the fitting function used to model the spacing distribution, which is the Wigner-Dyson distribution presented in Equation 2.4.
- `RMSE`: Computes the root mean squared error (RMSE) between the observed and expected data.
- `fitting_distribution`: Fits a spacing distribution to the fitting function, computes RMSE, and optionally prints the best-fit parameters and covariance matrix.
- `plot_fit`: Plots the histogram of the spacing distribution along with the fitted function.

All the functions are designed to be executed interactively in a Jupyter notebook. The results, including eigenvalue spectra, spacing distributions, and fit parameters, can be visualized using `matplotlib.pyplot` for plotting.

C. Code availability

All the data and the code for the data analysis can be found in this public Github repository: https://github.com/Kallo27/QIC/tree/main/Assignment3

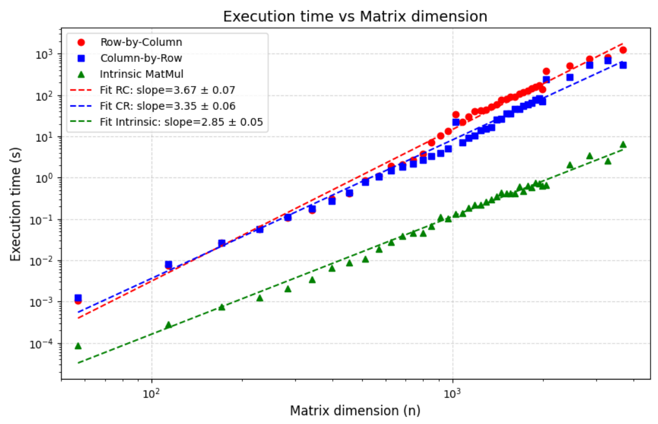

FIG. 1: Performance scaling of the three implemented methods for matrix-matrix multipication in Fortran90 (execution time vs matrix dimension).

4. RESULTS

A. Matrix multiplication

For the matrix-matrix multiplication performance analysis, we compare the execution times of the three implemented methods varying the input dimension of the matrix, n, in a range up to $n = 4096$. As previously said in Section 2 A, we can perform a linear fit to the data when plotted on a log-log scale, which is presented in Figure 1. As we can see, first of all the general behaviour is the one expected (we have indeed quite straight lines), both for the data and the fits; the results are gathered in Table I. As we can see the obtained slope are of the same order of magnitude as those expected: in particular, the highest is for the *row-by-column* implementation and the lowest is for the intrinsic `matmul` function. We can also notice that the *column-by-row* implementation is a bit faster than the *row-by-column* implementation, but in any case much slower than the intrinsic function, as expected.

Method	Slope	Error
Fit RC	3.67	0.07
Fit CR	3.35	0.06
Fit Matmul	2.85	0.05

TABLE I: Caption

B. Random matrices

Regarding random matrices, we studied in particular the normalized spacing distribution of the eigenvalues, $P(s)$, as described in Section 2 C. Figure 2 and Figure 3

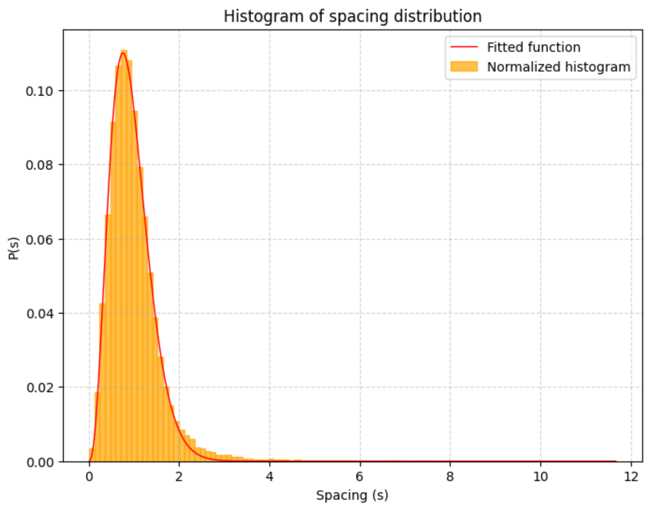

FIG. 2: Spacing distribution and fit for a random Hermitian matrix.

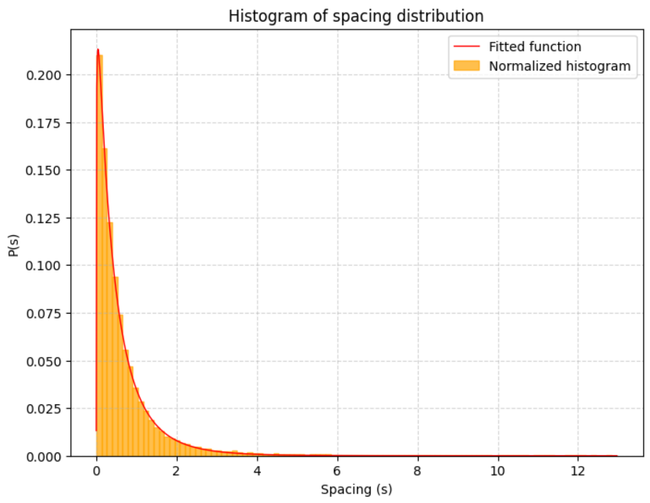

FIG. 3: Spacing distribution and fit for a random diagonal matrix.

represent the spacing distributions for random Hermitian and random diagonal matrices, respectively. As we can see, the general behaviour is the one expected: the Hermitian matrices present eigenvalues repulsion (and the distribution is peaked around a positive value), while the random diagonal matrices do not (and are peaked at 0). The result for the best parameters and the root mean squared error computed for the fit are gathered in Table II. As we can see for both distributions we got a small RMSE, which means a good agreement between the data and the expected behaviour. The difference between the two is that for the Hermitian matrices all the values of the parameters are near, meaning that the Wigner-Dyson distribution exhibits a behaviour similar to a Poissonian; on the other hand, for the diagonal matrices the value of b dominates over the others, leaving only the decreasing exponential contribution, as expected.

Matrix Type	a	alpha	b	beta	RMSE
Hermitian	1.1859	2.5624	-2.7378	1.3429	0.0009
Diagonal	0.6257	0.2328	-2.8598	0.6436	0.0005

TABLE II: Fitting parameters and RMSE for Hermitian and Diagonal matrices.

The same analysis has been conducted also removing the first eigenvalue, which in random matrices is often an outlier: the same result has been reached, which is not reported here in order to reduce repetitions. Some small variations in the parameter have been observed, always in agreement with the ones expected.

5. CONCLUSIONS

First of all, this assignment has demonstrated the computational complexity of matrix-matrix multiplication: by examining different implementations, we observed that intrinsic functions in Fortran, like `matmul`, offer substantial performance benefits by leveraging low-level optimizations provided by libraries like BLAS. In contrast, manually implemented loops, though educationally insightful, generally underperform for large matrices due to memory inefficiencies. Our study also highlights the utility of Python scripting in automating the process of parameter variation and data collection, enabling streamlined performance analysis across a range of matrix sizes.

Secondly, we studied random matrices, in particular exploring the eigenvalue spacing distributions of random Hermitian and diagonal matrices. Our findings confirmed that Hermitian matrices exhibit eigenvalue repulsion, with spacing distributions closely following the Wigner-Dyson model, while diagonal matrices lack this characteristic due to their inherent structural independence, as expected.

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. (Johns Hopkins University Press, 2013).
[2] Fortran-Lang, *Fortran Programming Language Community*, https://fortran-lang.org/.
[3] Strang, Gilbert. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2016.
[4] Python Software Foundation. "subprocess — Subprocess management." *Python Documentation*, Python 3.12.0, 2023. Available: https://docs.python.org/3/library/subprocess.html.
[5] M. L. Mehta, *Random Matrices*, 3rd ed. (Elsevier/Academic Press, 2004).