

# Quantum Information and Computing

## Assigment 3 Report

Andrea Turci

November 2024

### Scaling of the Matrix-Matrix Multiplication

#### 1 Introduction

Matrix multiplication is one of the most fundamental operations in computer science and mathematics, and it is widely used in various fields, including machine learning and quantum computing. The problem addressed in this assignment is to analyze the performance scaling of matrix-matrix multiplication algorithms. Specifically, we aim to explore how the execution time of matrix multiplication changes as a function of matrix size,  $N$ , for different multiplication methods. I will compare different methods and algorithms, fitting the scaling of execution time as a function of the matrix size and compare these to theoretical performance models and expectations.

In order to achieve this, a Python script has been written, that varies the matrix size  $N$  in the multiplication between two user-specified values,  $N_{min}$  and  $N_{max}$ , and that launches the program, compiling and running it.

Additionally, the code is written in a proper way to store the execution time results in separate files according to the parameters used in the simulation, such as the multiplication method used. Finally, the results are plotted and visualized, from which it is possible to highlight the matching with respect to the expected scaling behavior.

#### 2 Theoretical Framework

As mentioned previously, matrix multiplication is a fundamental operation in many computational fields. The multiplication of two matrices  $A$  (size  $N \times N$ ) and  $B$  (size  $N \times N$ ) results in a matrix  $C$  (also  $N \times N$ ) where each element is computed as a dot product of corresponding rows and columns:

$$C_{ij} = \sum_{k=1}^N A_{ik} B_{kj} \quad (1)$$

However, the way this dot product is computed can vary. Different algorithms use different strategies for computing matrix multiplication, and the efficiency of each also depends on factors such as memory access patterns, cache usage, and computational overhead.

While the classical matrix multiplication method has a time complexity of  $O(N^3)$ , there are variations in the way multiplication is carried out, and each approach can be optimized differently depending on the underlying hardware.

## 2.1 Row-by-Column Matrix Multiplication

In the Row-by-Column method, sometimes also called naive method, the elements of the result matrix  $C$  are computed by taking the dot product of rows of matrix  $A$  and columns of matrix  $B$ . This is the classical approach used in many basic matrix multiplication implementations. Remembering this once again, for each element  $C_{ij}$  of the resulting matrix  $C$ , we compute:

$$C_{ij} = \sum_{k=1}^N A_{ik} B_{kj} \quad (2)$$

where  $A_{ik}$  is the element of matrix  $A$  in row  $i$  and column  $k$  and  $B_{kj}$  is the element of matrix  $B$  in row  $k$  and column  $j$ .

This method requires iterating through each element of the resulting matrix  $C$ , which has  $N^2$  elements, and computing the sum of products for each element, which requires  $N$  multiplications and additions per element. Hence, the time complexity of the Row-by-Column method is  $O(N^3)$ .

## 2.2 Column-by-row Matrix Multiplication

The Column-by-Row method is an alternative method used to perform the matrix multiplication, where the difference occurs in the order the multiplication is performed, which from a coding point of view has an effect on the priority of the loops performed, whether by row or by column. That being said, if I want to retrieve the element  $C_{ij}$  of matrix  $C$  we compute it using this formula:

$$C_{ij} = \sum_{k=1}^N A_{ki} B_{kj} \quad (3)$$

where  $A_{ki}$  is the element of matrix  $A$  in row  $k$  and column  $i$  and  $B_{kj}$  is the element of matrix  $B$  in row  $k$  and column  $j$ . Notice here the presence of  $A_{ki}$  instead of  $A_{ik}$ . Taking the same considerations as for the previous method, we expect a time complexity still in the order of  $O(N^3)$  for this method. However, the primary advantage of the Column-by-Row method is that it accesses memory in a more cache-friendly manner, since the elements from matrix  $A$ , which are arranged in columns, are accessed in a more sequential manner, improving memory locality.

## 2.3 Optimized Matrix Multiplication - MATMUL intrinsic function

The MatMul method refers to the use of optimized algorithms for matrix multiplication, such as the widely used BLAS (Basic Linear Algebra Subprograms) implementation.

In this optimized approach, the multiplication process is performed using highly efficient routines that take into account for example the cache locality. In particular, the algorithm ensures that data is accessed in a manner that maximizes cache utilization, reducing memory access latency; also, from the parallelization point of view, the algorithm is often parallelized, taking advantages of multiple cores. Moreover, larger matrices are split into smaller blocks, allowing them to fit into the processor's cache and reducing the number of cache misses.

From the point of view of time complexity, the Matmul method can achieve performances close to  $O(N^2)$  in practice, reducing significantly the time for large matrices. However, the theoretical complexity remains cubic,  $O(N^3)$ , but practical implementations can outperform this bound due to various optimizations.

For example, Strassen's Algorithm, an optimized matrix multiplication algorithm, reduces the number of scalar multiplications needed from  $O(N^3)$  to  $O(N^{2.81})$ , offering a significant improvement for very large matrices.

So, as a conclusion, the theoretical CPU time expected for the Matmul method is expected to

have a polynomial dependence much more improved than the other methods.

The primary goal of this exercise in this report is to compare the empirical execution times for these different methods and validate their scaling behavior against the theoretical models.

## 2.4 Compilation Flags

In this report, optimization flags are leveraged as a method to enhance computational performance by controlling the level of code optimization during compilation. Specifically, the `gfortran` compiler provides a range of optimization flags designed to improve execution speed and efficiency by varying degrees of code transformation and optimization. Each flag level engages progressively advanced optimization techniques, balancing between compile time, runtime efficiency, and resource utilization.

The `-O1` flag is the most fundamental level of optimization and primarily targets essential improvements that do not significantly impact compilation time. This flag enables basic optimization processes, such as eliminating redundancies, minimizing memory accesses, and simplifying expressions where possible. While this level of optimization can improve performance relative to unoptimized code, it does not deeply alter code structure, making it suitable for quick compilation and basic performance gains without altering program behavior.

The `-O2` flag represents a moderate level of optimization, making it an optimal choice in many scenarios where performance is prioritized without drastically increasing compilation complexity. At this level, the compiler performs more comprehensive optimizations, particularly focusing on loop efficiency and memory access patterns. For instance, `-O2` applies optimizations that reduce the frequency of memory access within loops, unrolls loops when advantageous, and applies function inlining selectively to reduce overhead. As a result, the `-O2` flag is often a practical balance, yielding substantial runtime improvements while avoiding the potential drawbacks of higher optimization levels, such as increased compilation time or unintended side effects from aggressive code transformations.

The `-O3` flag is the most aggressive optimization level, introducing additional transformations, including automatic vectorization and more extensive function inlining. This level also applies advanced loop transformations, aiming to maximize parallelism and leverage modern CPU architectures more effectively. With `-O3`, the compiler attempts to exploit instruction-level parallelism, which can significantly accelerate performance, particularly in computationally intensive programs involving heavy numerical calculations or matrix operations. However, these optimizations may increase memory usage and can alter the behavior of floating-point calculations due to aggressive approximations, making `-O3` best suited for scenarios where maximum speed is prioritized over factors such as compilation time or strict precision requirements.

By examining and comparing the performance outcomes for each of these optimization flags, this study aims to assess the impact of varying levels of optimization on matrix computation efficiency. The analysis highlights the trade-offs associated with each flag level and provides insight into how different optimization techniques influence both runtime and computational accuracy, particularly in matrix-intensive applications.

## 3 Methodology

The first goal of analyzing the scaling of the matrix multiplication with different methods is to write and create a Python script in order to automate the process of compiling, running and collecting the results from the Fortran code.

In particular, as will be analyzed in later sections, the code which performs the matrix multiplication is inserted into Fortran `.f90` files, which contain all the programs, modules and subroutines necessary to perform the multiplication and store the results in text files, allowing for a subsequent analysis. Instead of asking for the input parameters in command-line or, even

worse, defining them in-line in the code, the Python script reads the input parameters from a text file, compiles and executes the Fortran files, handling errors during the compilation.

### 3.1 Code Implementation and Description

The code is organized in this way:

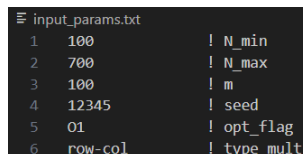
- `matrix_prog.f90`: Fortran file that contains the program which calls the other functions contained in the next file;
- `matrix_module.f90`: Fortran file which contains subroutines for matrix multiplication and writing to files;
- `script.py`: Python script which automate the compilation and execution of the Fortran files reading the input parameters from a text file;
- `input_params.txt`: text file which contains the input parameters to provide in the execution of the program.

That said, I begin to describe the code by starting with the files that are used first. For this reason, let us analyze the text file for the input parameter.

The parameters that are asked as input to make the program work are:

- `N_min`: that is the minimum matrix size for which the matrix multiplication scaling is tested. It must be an integer number and non-negative.
- `N_max`: that is the maximum matrix size for which the matrix multiplication scaling is tested. It must be an integer number greater than `N_min` and non-negative.
- `m`: which controls the increment size for the matrix dimension in each test. It must be smaller than `N_max` and non negative.
- `seed`: which ensures reproducibility of random matrix entries. It must be an integer value.
- `opt_flag`: which specifies the compiler optimization level. It is a variable of type character, and the only possible values for it can be `O1`, `O2` or `O3` depending of the desired flag to use for the compilation.
- `type_mult`: which chooses the multiplication methods to evaluate. It is a variable of type character and the only possible values for it can be `row-col`, `col-row`, `matmul` or `ALL` depending on the desired multiplication method to apply. In particular, the value `ALL` applies all the three multiplication methods in other to speed up the process when one wants to compare all the three methods at once.

According to this, the `input_params.txt` file will look like this:



```
input_params.txt
1 100      ! N_min
2 700      ! N_max
3 100      ! m
4 12345    ! seed
5 O1       ! opt_flag
6 row-col  ! type_mult
```

**Figure 1:** Text file for input parameter

Moving on to the next file, `matrix_prog.f90` contains instead the program which defines the five above-mentioned variables and reads them. The module makes use of the debugger module, in such a way to perform checkpoints and error handling during the compilation and execution, and of the `matrix_mult` module, which contains all the needed subroutines and

which is defined in the `matrix_module.f90` file. In particular, the program has the following structure, and performs the subroutine `perform_multiplication`:

```

57
58
59 program matrix_multiplication_performance
60   use debugger      ! Import debugging module for checkpoints
61   use matrix_mult    ! Import matrix multiplication module
62   implicit none
63
64   ...
65
66   ! Main subroutine to execute matrix multiplications based on input parameters
67   call perform_multiplications(min_size, max_size, step, seed, opt_flag, type_mult)
68
69 end program matrix_multiplication_performance

```

**Figure 2:** Program for matrix multiplication

Afterwards, it is possible to start describing the file `matrix_module.f90` which contains all the subroutines necessary to perform the matrix matrix multiplication and to write the results in a text file in a proper format.

The first subroutine we encounter is called `prepare_output_file`. This subroutine prepares the output filename based on user parameters, and checks if the file exists (if the file does not exist, it creates a new file and writes a header). The most important thing to mark is the way the output file is named: each output file is defined according to the parameters chosen for the simulation using the format

`<type_mult>_size-<N_min>-<N_max>_step-<m>_flag-<opt_flag>.dat`.

An example of an output file can be:

```

row-col_size_100-700_step_100_flag_O1.dat

```

**Figure 3:** Example of output text file

This is done in order to immediately recognize the parameters used in a given simulation. Additionally, the subroutine writes the header according to the multiplication method used: apart from an always present column `Size` the header will contain the names of the columns for the given method used, as the following code explains:

```

if (type_mult == "ALL") then
  write(20, '(A)') ' Size Explicit(i-j-k) Column-major(i-k-j) MATMUL'
else if (type_mult == "row-col") then
  write(20, '(A)') ' Size Explicit(i-j-k)'
else if (type_mult == "col-row") then
  write(20, '(A)') ' Size Column-major(i-k-j)'
else if (type_mult == "matmul") then
  write(20, '(A)') ' Size MATMUL'
end if

```

**Figure 4:** Code to write the header into output text files

Then, we encounter the subroutine `perform_multiplication`, which is the main subroutine of this module, since it calls all the other subroutines, and which is also the one called by the `matrix_prog.f90` file, as it can be seen in Figure 2.

The first thing this subroutine does is to print a message, using the `debugger` module, which notifies and reports that the matrix multiplication process has started. Secondly, it performs the error handling of the parameters, checking whether or not the input parameters lie in the correct range (explained before) and are of the correct type.

Furthermore, it calls the subroutine `prepare_output_file` explained before, and performs the seed management for random number generation calling the function `random_seed` and

according to the seed value given as input. All this can be seen in the following image:

```
call checkpoint_real(debug=.TRUE., msg='Beginning matrix multiplication process.')
if (max_size <= 0 .or. step <= 0 .or. step >= max_size) then
  print*, "Error: Invalid matrix size or step configuration."
  return
end if

call prepare_output_file(filename, type_mult, min_size, max_size, opt_flag, step, seed)

call random_seed(size=m)
allocate(seed_array(m))
seed_array = seed
call random_seed(put=seed_array)
```

**Figure 5:** *Matrix multiplication Module*

Afterwards, the subroutine loops over the matrix sizes from  $N_{\min}$  to  $N_{\max}$  with the specified step  $m$ , initializes the two matrices  $A$  and  $B$  with random values and performs the selected multiplication method according to the one given as input. Indeed, the CPU time taken to perform the multiplication is computed, and, using the debugger module, a checkpoint is used to print the CPU time for this iteration in the loop. The following image helps understanding this step, which is indeed dependent on the used multiplication method:

```
if (type_mult == "ALL" .or. type_mult == "matmul") then
  ! MATMUL intrinsic method
  call cpu_time(start_time)
  c_intrinsic = matmul(A, B)
  call cpu_time(end_time)
  time_matmul = end_time - start_time
  call checkpoint_real(debug = .TRUE., verbosity = 2, msg = 'Time taken for intrinsic MATMUL', var1 = time_matmul)
end if
```

**Figure 6:** *Calculation of CPU time according to the used method*

The subroutines which perform the matrix multiplication following the row-by-column and column-by-row methods are straightforward and not reported in this context for sake of simplicity.

As a final thing, the subroutine writes the results into the correct output file, based on the selected method:

```
if (type_mult == "ALL") then
  write(20, '(I6, 3X, F12.6, 3X, F12.6, 3X, F12.6)') i, time_explicit, time_column, time_matmul
else if (type_mult == "row-col") then
  write(20, '(I6, 3X, F12.6)') i, time_explicit
else if (type_mult == "col-row") then
  write(20, '(I6, 3X, F12.6)') i, time_column
else if (type_mult == "matmul") then
  write(20, '(I6, 3X, F12.6)') i, time_matmul
end if
```

**Figure 7:** *Writing into output text file of the results*

This concludes the discussion about the Fortran files and opens the one about the Python script.

The script file consists of several functions, all of them called inside the main execution block. The first one is the function `read_parameters` which reads parameters from the input file called `input_params.txt` (whose template is shown in Figure 1) where for each line it ignores empty lines and comments, and convert values to appropriate types. If the values for the integer variable is not valid, or the input parameter file is not found, the function throws an exception.

```
def read_parameters(filename):
    params = {}
    try:
        with open(filename, 'r') as f:
            for line in f:
                if line.strip() and not line.strip().startswith('!'):
                    parts = line.split('!', 1)
                    value = parts[0].strip()
                    name = parts[1].strip() if len(parts) > 1 else ''
                    params[name] = value

        return {
            'Nmin': int(params.get('N_min', 0)),
            'Nmax': int(params.get('N_max', 0)),
            'm': int(params.get('m', 0)),
            'seed': int(params.get('seed', 0)),
            'opt_flag': params.get('opt_flag', ''),
            'type_mult': params.get('type_mult', '')
        }

    except ValueError:
        print('Please enter valid integers for Nmin, Nmax, and m.')
        sys.exit(1)
    except FileNotFoundError:
        print('Input parameters file not found.')
        sys.exit(1)
```

**Figure 8:** *Function to read the parameters from input file*

Afterwards, the main block performs the error handling of the input variable to check their correctness and that the character variables are initialized to values within the possible ones.

Then, the script compiles the files `debugger.f90` and `matrix_module.f90` using the function `compile_module` which uses the library `subprocess` to run them using the compiler `gfortran` and throws an exception if an error during the compilation occurs:

```
def compile_module(module_file):
    """Compiles a Fortran module using gfortran."""
    try:
        print(f'Compiling {module_file}...')
        subprocess.run(['gfortran', module_file, '-c'], check=True)
        print('Compilation successful!')
    except subprocess.CalledProcessError as e:
        print(f'Error during compilation: {e}')
        return False
    return True
```

**Figure 9:** *Function to compile the Fortran modules*

In a similar way it is called the function `compile_fortran` which compiles the Fortran file `matrix_prog.f90` as source file, with additional modules that are the object files `debugger.o` and `matrix_module.o`.

Finally, the function `run_executable` is used to run the compiled Fortran executable `matrix_prog.x` and which throws an exception if an error during the execution happens thanks to the `subprocess` library as well. The structure of the function is the following:

```
def run_executable(exec_name, input_data):
    try:
        params = {
            ...
        }

        # Format the output message
        formatted_input = ', '.join(f'{key}={value}' for key, value in params.items())
        print(f'Running {exec_name} with input: {formatted_input}')

        # Run the executable with the input data
        result = subprocess.run([f'./{exec_name}'], input=input_data, text=True, capture_output=True, shell=True, check=True)
        print('Execution successful.\n')

        # Return the output from the program
        return result.stdout
    except subprocess.CalledProcessError as e:
        print(f'Error during execution: {e}')
        return None
```

**Figure 10:** *Script function to run the fortran executable*

### 3.2 Running Procedure and Output

Having previously described the code, the first thing to do is to choose the input parameters to run the simulation and to fill in the `input_params.txt` file.

Secondly, one has to write on terminal the following: `python script.py`.

On the terminal it will appear an output similar to the following

```
Compiling debugger.f90...
Compilation successful!
Compiling matrix_module.f90...
Compilation successful!
Compiling matrix_progr.f90 with additional modules: ('debugger.o', 'matrix_module.o')...
Compilation successful!
Running matrix_progr.x with input: Nmin=100, Nmax=700, n_points=100, seed=12345, flag=01, type_mult=row-col
Execution successful.

Finished saving on file, named row-col size 100-700 step 100 flag 01 12345.dat
```

**Figure 11:** *Terminal Output printings*

which, thanks to the checkpoints defined in the `debugger` module, summarizes the parameters used in the simulation and notifies the correctness of the compilation, execution and writing to file, as well as explicitly reporting the name of the output file.

On the other hand, the output file will look like this:

Size	Explicit(i-j-k)
100	0.007399
200	0.040099
300	0.130527
400	0.302734
500	0.655628
600	1.111895
700	1.820994

**Figure 12:** *Example of the content of output files*

showing that the both the header and the results have been correctly written, allowing for a further analysis.

## 4 Results

First of all, the CPU time for the three different methods have been computed, keeping the same seed, for all the 3 possible flags, and the results are shown in Figure 13, where the trends are separated according to the matrix multiplication method used.

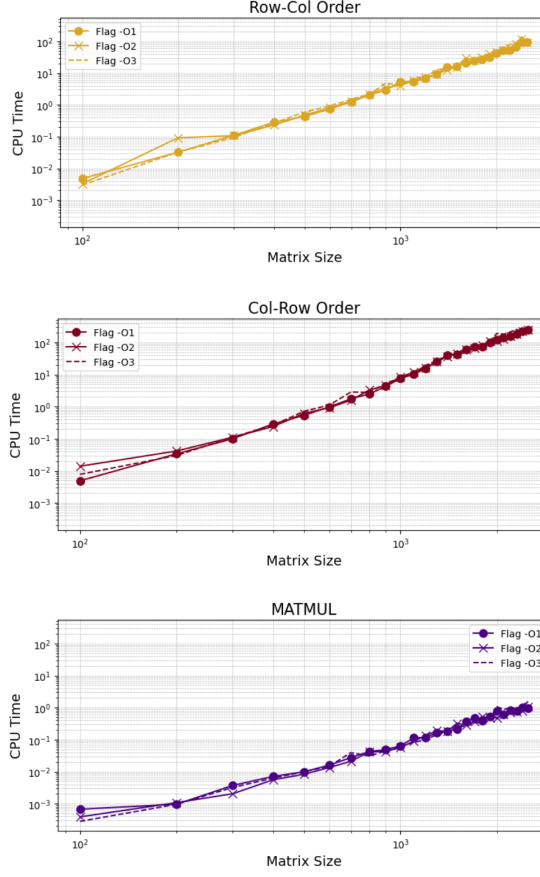
In particular, the axis are put in log/log scale in such a way to highlight the expected polynomial behaviour: a linear trend emerges in log/log scale, suggesting and highlighting the presence of a polynomial trend of the CPU time as a function of the matrix size, which needs to be further analyzed.

Additionally, it is possible to notice that the flags applied in the compilation procedure doesn't affect so much the performance, allowing us to consider the results applying the flag 02 in the following analysis.

The next step is to perform a polynomial fit of the results obtained with different methods, keeping the same input parameters and using the compilation flag 02. The fit is performed defining a polynomial function which two parameters: the amplitude and the exponent. Indeed, these are the two parameters which allow us to make a complete and exhaustive comparison between the 3 different methods.

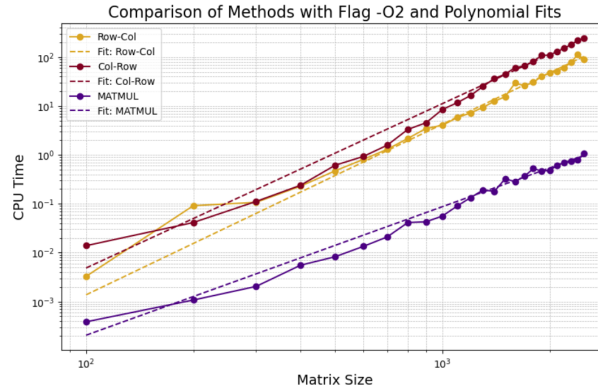
The results are shown in Figure 14, where the solid line represents the CPU data versus the matrix size, while the dashed line represents the fit for the associated method. Again both the





**Figure 13:** CPU time versus the Matrix Size for different multiplication methods and different flags

axis are put in logarithmic scale, in such a way that the polynomial fit appears as a straight line where the slope is identical to the exponent of the polynomial. Additionally, the results of the fit are reported in Table 1.



**Figure 14:** CPU time versus the matrix size of different methods (flag O2) and correspondant fit

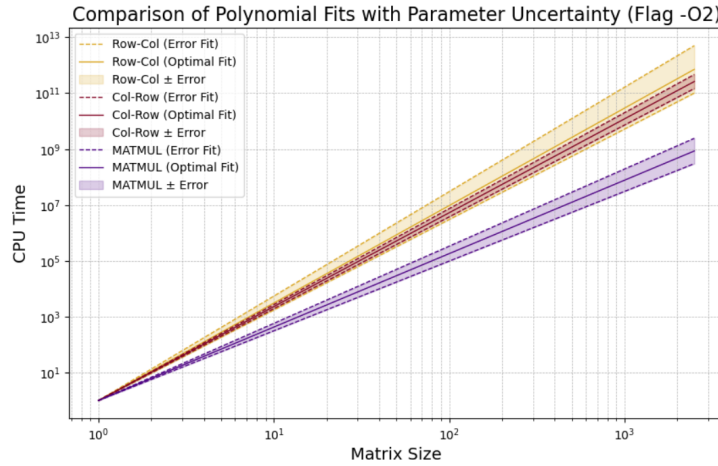
Method	Exponent ( $r$ )	Chi-Squared ( $\chi^2$ )
Row-by-Column	$3.489 \pm 0.250$	13.030
Column-by-Row	$3.361 \pm 0.074$	10.324
MATMUL	$2.629 \pm 0.134$	0.125

**Table 1:** Table of the results of the polynomial fit for the different multiplication methods

As it is possible to notice, the Column-by-Row method works slightly better than the Row-by-Column one, as expected from the Theoretical Framework Section, having both a time complexity in the order of  $O(N^3)$ ; indeed, from Equation 3, due to the presence of  $A_{ki}$  instead of  $A_{ik}$  the Column-by-Row method accesses memory in a more cache-friendly since the elements from matrix  $A$  are accessed in a more sequential manner.

Additionally, the results show that the the MatMul intrinsic function of Fortran works significantly better than the other two, with a scaling time complexity in the order of  $O(N^{2.629})$ , as expected from the Theoretical Framework section.

A better way to visualize the polynomial scaling of the time complexity is visualized in Figure 15, where also the errors associated to the exponents are taken into account.



**Figure 15:** Comparison of the fit results for the different multiplication methods with the parameter uncertainty

## 5 Conclusions

This exercise analyzed the computational scaling of different matrix multiplication methods — Row-by-Column, Column-by-Row, and the optimized MatMul intrinsic function—across various matrix sizes and compiler optimization levels. Each method revealed unique performance characteristics. The Row-by-Column and Column-by-Row methods, both with time complexities of approximately  $O(N^3)$ , performed as expected, with the Column-by-Row approach showing slight efficiency gains due to better memory access patterns.

The MatMul intrinsic function significantly outperformed the other methods, achieving practical scaling closer to  $O(N^{2.63})$ , which aligns with theoretical predictions for optimized algorithms in specific scenarios. This improvement highlights the effectiveness of intrinsic functions, which leverage advanced optimizations and parallel processing where possible.

Overall, this study demonstrates the importance of choosing appropriate matrix multiplication methods and compiler optimizations when scaling large matrix computations, as these decisions markedly impact computational efficiency.

# Random Matrix Theory

## 1 Introduction

In this exercise, we delve into the statistical analysis of eigenvalue spacings in random matrices, specifically focusing on Hermitian matrices with complex entries and real diagonal matrices. The aim of this exercise is thus to explore the behavior of eigenvalues for these random matrices and to understand the distribution of normalized eigenvalue spacings. Such analyses are fundamental in random matrix theory (RMT), a field that models the statistical properties of eigenvalues in large, complex systems. RMT is widely used in various disciplines, including physics, mathematics, and quantum computing, as it offers insights into the spectral characteristics of systems such as atomic nuclei, chaotic Hamiltonians, and quantum systems with many-body interactions. For this assignment, we study the probability distribution function  $P(s)$ , which describes the normalized spacing  $s$  between adjacent eigenvalues. By comparing  $P(s)$  distributions for both Hermitian matrices and real diagonal matrices, we aim to discern patterns that may reflect underlying statistical properties unique to each type. This distribution reveals essential insights, such as the degree of repulsion between eigenvalues, which is a characteristic that differentiates complex systems with varying levels of randomness and correlation.

## 2 Theoretical Framework

Random matrix theory provides a mathematical framework for understanding the statistical distribution of eigenvalues in large, complex systems. It was initially developed by physicist Eugene Wigner in the 1950s to model the spectra of heavy atomic nuclei, which are inherently chaotic and involve numerous interacting particles. Over time, RMT has been applied to a wide array of fields, including quantum mechanics, number theory, and complex systems. At the core of RMT is the study of eigenvalue statistics, where eigenvalue spacings reveal information about the underlying physical or mathematical structure of the system.

For this exercise, I consider two specific matrix classes:

- **Random Hermitian Matrices:** These are complex matrices  $A$  with the Hermitian property  $A_{ij} = A_{ji}^*$ , meaning the matrix is equal to its conjugate transpose. For Hermitian matrices, all eigenvalues are real, and their distribution exhibits a level of repulsion indicative of underlying quantum chaotic systems. The eigenvalue repulsion observed in these matrices aligns with the Gaussian Unitary Ensemble (GUE) in random matrix theory.
- **Real Diagonal Matrices:** These are diagonal matrices where entries are real random numbers with no correlation. In this case, the eigenvalues are uncorrelated, and we expect no significant repulsion between them. Thus, real diagonal matrices serve as a reference for analyzing eigenvalue distributions without inherent correlations or structure.

The eigenvalue spacings between adjacent eigenvalues for a matrix  $A$  are defined by:

$$\Lambda_i = \lambda_{i+1} - \lambda_i \quad (4)$$

where  $\lambda_i$  represents the  $i$ -th eigenvalue of the matrix in ascending order. The normalized spacing  $s_i$  is obtained by dividing each spacing  $\Lambda_i$  by the mean spacing  $\bar{\Lambda}$ , defined as:

$$s_i = \frac{\Lambda_i}{\bar{\Lambda}} \quad (5)$$

where  $\bar{\Lambda}$  is the average of  $\Lambda_i$  values across all eigenvalues.

The probability distribution function  $P(s)$ , describing the distribution of  $s$ -values, is crucial for

differentiating matrix classes. Indeed for random Hermitian matrices,  $P(s)$  typically exhibits level repulsion, meaning small spacings between eigenvalues are less likely, leading to a probability density with a peak away from zero. On the other hand, for diagonal matrices with real entries, eigenvalues do not exhibit repulsion, and we expect a distribution where small spacings are more probable, often resembling a Poisson distribution.

We fit the observed  $P(s)$  to a theoretical model given by:

$$P(s) = as^\alpha e^{bs^\beta} \quad (6)$$

where  $a$ ,  $b$ ,  $\alpha$  and  $\beta$  are fitting parameters, each capturing distinct characteristics of the eigenvalue distribution.

The function given by Equation 6 is chosen as a flexible fit for the spacing distribution because it provides a versatile framework for capturing the distinct spacing behaviors of different types of matrices, being able to capture both repulsive and non-repulsive behaviors in eigenvalue spacings, depending on the values of  $\alpha$  and  $\beta$ .

In particular, thanks to the power-law term  $s^\alpha$  and exponential term  $e^{bs^\beta}$ , it is possible for the function to take a wide range of shapes: by adjusting these parameters, the function can approximate both Gaussian-like shapes and Poisson-like shapes.

Here's how different settings correspond to common distributions in random matrix theory, depending on the different parameters  $a$ ,  $b$ ,  $\alpha$  and  $\beta$ :

- Wigner-Dyson Distribution (for Hermitian Matrices): it has  $\alpha = 1$  and  $\beta = 2$ . In this way, the linear term in  $s$  (with  $\alpha = 1$ ) combined with the squared term in the exponential (with  $\beta = 2$ ) creates a curve that rises for small  $s$ , peaks at a positive  $s$  value, and decays smoothly.
- Poisson Distribution (for Real Diagonal Matrices): it has  $\alpha = 0$  and  $\beta = 1$ . In this way, the probability density is high at small spacings, and follows an exponential decay.
- Intermediate Distributions:  $\alpha$  and  $\beta$  can be varied to model intermediate cases.

## 2.1 Quantum Random Number Generation

The report focuses also on the importance of generating random numbers through quantum principles, which is essential for ensuring the authenticity and unpredictability of randomness itself. In the context of this assignment, random numbers are generated by observing the time intervals between photon arrivals from a coherent light source, a process that follows an exponential distribution typical of a Poisson process. The data are collected from an experiment run in the course “Quantum Optics and Laser” (whose Report can be found [here](#)).

This type of distribution implies that each photon detection is independent of previous events, a property essential for producing a truly random sequence of numbers that is unaffected by classical deterministic systems. This intrinsic randomness makes quantum processes ideal for constructing quantum random number generators (QRNGs), as they guarantee the unpredictability required in applications demanding high levels of security and reliability in randomness. Within this analysis, quantum random numbers are used to study the spectral characteristics of generated matrices, allowing a comparison of the behavior of matrices generated from purely quantum mechanisms with that of matrices generated by classical methods. More about the Quantum Random Number Generation using photon counts can be found in Appendix 6

## 2.2 Sparse Matrices

Sparse matrices, which feature predominantly zero-valued elements with only a minority of non-zero entries, are highly beneficial in simulations and computations involving large matrices, where only a small fraction of elements contributes to the final result. This characteristic renders

sparse matrices highly efficient in terms of memory and computational time, as they significantly reduce the number of operations required and the volume of data to manage, especially when compared to dense matrices.

In the specific case of this study, creating sparse Hermitian matrices requires additional steps, such as randomly selecting indices for the non-zero entries, to respect the Hermitian property of complex conjugate symmetry. However, this requirement proved to be a limitation in practice, as the increased computational load associated with the index selection and assignment of non-zero complex values resulted in longer execution times than those required by conventional methods, diminishing the anticipated advantages of the sparse approach. Thus, as we will see, in this context the use of sparse matrices proved less effective, as it failed to provide the expected performance gains compared to traditional matrix generation methods.

### 3 Methodology

To analyze the eigenvalue spacing distributions, we follow a structured methodology that includes matrix generation, eigenvalue computation, spacing normalization, accumulation across realizations, and statistical fitting.

In particular, for this exercise I abandoned Fortran and used only Python to define all the functions and run the simulations.

#### 3.1 Code Implementation and Description

The main part of the code is organized in this way:

- `functions.py`: pythonic file which contains all the functions that are used to analyze spacing distribution, generate matrices, compute the eigenvalues and spacing normalization, and perform the fit.
- `hermitian_matr.ipynb`: jupyter notebook which imports the functions defined in `functions.py` and makes use of those functions to run the simulations.

That said, I begin to describe the code by starting with the files that are used first. For this reason I start describing the `functions.py` file and analyzing the input parameters that are needed to run the simulation.

- `N`: integer variable representing the dimension of the matrices;
- `num_matrices`: integer variable representing the number of matrices that are used in the simulation, for averaging the results and having better statistics;
- `seed`: integer variable representing the seed used in the random process for the initialization of the matrices;
- `N_bins`: integer variable representing the number of bins to use in the histogram;
- `flag`: a list variable containing the distribution flags, according to the type of random generation one requires (see later);
- `type`: a string variable for the matrix type: it can be either "hermitian", "diagonal" or "sparse", according to the type of matrix one wants to generate.

To begin with, we consider `num_matrices = 1000`, since we want to accumulate values from different random matrices, where for each matrix we change the seed, in such a way to obtain more significant and random results and a better statistics.

According to the type of matrix we want to generate, we have different functions:

- `generate_random_hermitian_matrix` for random hermitian matrices;
- `generate_real_diag_matrix` for real diagonal matrices;
- `generate_sparse_random` for random sparse matrices.

Despite these 3 functions generate different matrices, they all are related to the problem of generating random numbers; for this reason they all have the same template and structure. For each of the 3 functions we can choose the `flag` parameter between values 0,1,2, choosing in this way the type of random generation:

- `flag=0`: I randomly generate the entries using standard normal distribution with `mean=0` and `std=1`.
- `flag=1`: I randomly generate the entries between -1 and 1.
- `flag=2`: I randomly sample the values from a list of quantum random generated values coming from the [Arecchi Experiment](#) (performed in the course Quantum Optics and Laser, whose report can be found [here](#)). As said before, the experiments consists in the photon detection of a coherent light, where the arrival of photons follows a Poissonian process, where the time intervals between subsequent photon detections are exponentially distributed. This exponential distribution ensures that the photon detection process is random and that each detection event is independent of previous events. More information about the quantum random number generation process, the code relative to it and the degree of randomness that this process has can be found in [Appendix 6](#).

Indeed, one of the goals of this exercise is to compare the performances and the results of these 3 different methods of generating numbers.

According to the desired type of matrix, the eigenvalues and the normalized spacing are computed in this way:

```
def compute_normalized_spacings(N, seed, flag, type, density=0.1):
    if (type == "hermitian"):
        A = generate_random_hermitian_matrix(N, seed, flag)
        eigenvalues = np.linalg.eigh(A)[0] # Sorted real eigenvalues
    elif (type == "diagonal"):
        A, eigenvalues = generate_real_diag_matrix(N, seed, flag)
        sorted_indices = np.argsort(eigenvalues)
        eigenvalues = eigenvalues[sorted_indices]
    elif (type == "sparse"):
        A, eigenvalues = generate_sparse_random(N, seed, density, flag)
        sorted_indices = np.argsort(eigenvalues)
        eigenvalues = eigenvalues[sorted_indices]
    else:
        print("Invalid type for matrix generation")

    # Compute spacings and normalize by mean spacing
    spacings = np.diff(eigenvalues)
    avg_spacing = np.mean(spacings)
    normalized_spacings = spacings / avg_spacing
    return A, eigenvalues, normalized_spacings
```

**Figure 16:** Function to compute the matrix and normalized spacings according to the type value

Looping up to the number of matrices `num_matrices`, changing the seed for each of them, the function `calculate_Ps_distribution` is then called and all the normalized spacings are appended into a unique list.

Afterwards, the function `fitting` performs the fit of the observed distribution with the target function of Equation 6, yielding as output the best parameters  $a$ ,  $b$ ,  $\alpha$ ,  $\beta$  and the  $\chi^2$  value.

Finally, the function `results` computes the distribution  $P(s)$  and the fit of the observed data looping over the different flags taken in input, and selecting only the best results, that is the one having the smallest  $\chi^2$  value, as it is possible to see in the next Figure:

```

for i in flag:
    spacings = calculate_Ps_distribution(N, num_matrices, i, type)

    print(f"Fitting for flag {i}...")
    a, b, alpha, beta, chi_square, fitted_P_s = fitting(N_bins, spacings, i, type)

    if chi_square < best_chi_square:
        best_chi_square = chi_square
        best_flag = i
        best_params = (a, b, alpha, beta)
        best_fit_P_s = fitted_P_s
        best_spacings = spacings

```

**Figure 17:** code part that computes the distribution  $P(s)$  and the its fitting, keeping the best parameters

Finally, the results are plotted only for the best flag in the simulation, so basically for the process of random generation that provides the best  $\chi^2$  value.

## 4 Results

First of all, the hermitian matrices are generated in order to analyze the distribution  $P(s)$ , using this set of parameters:

- $N = 1000$
- $\text{num\_matrices} = 20$
- $N\_bins = 100$
- $\text{flag} = \text{np.array}([0,1])$
- $\text{type} = \text{"hermitian"}$

Calling the function results one gets the following output:

```

Fitting for flag 0...
# Generate using standard normal distribution
Fitted parameters:
a = 16.7227, b = -3.0093, alpha = 2.7378, beta = 1.2988
Chi-square: 0.00016133634015801652

Fitting for flag 1...
# Generate values between -1 and 1
Fitted parameters:
a = 0.0681, b = 1.0255, alpha = -0.4951, beta = 0.0000
Chi-square: 0.25280065856452455

Best flag: 0
Best chi-square value: 0.00016133634015801652
Best parameters: a = 16.7227, b = -3.0093, alpha = 2.7378, beta = 1.2988

```

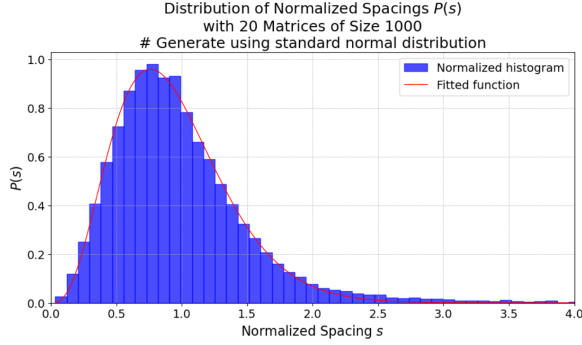
**Figure 18:** Terminal Output printings

while the plot for the best flag (0) is represented in Figure 19, where the best values of the fit are summarized in Table 2.

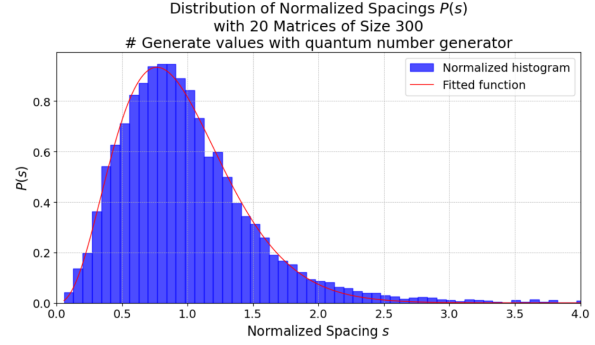
First of all, it is possible to see the presence of a Bell, which does not perfectly resemble the Wigner-Dyson distribution, but which highlights level repulsion typical for Hermitian matrices being a probability density with a peak away from zero.

Afterwards, the simulation is run keeping the same parameters but with  $\text{flag}=2$  in order to test the randomness of the quantum-generated numbers with the Poissonian process of photon detection. In this scenario the maximum matrix size is about  $N \simeq 300$  because of the restricted

number of photons counts that we have in the dataset. The results are depicted in Figure 20 and the best values of the fit are summarized in Table 2.



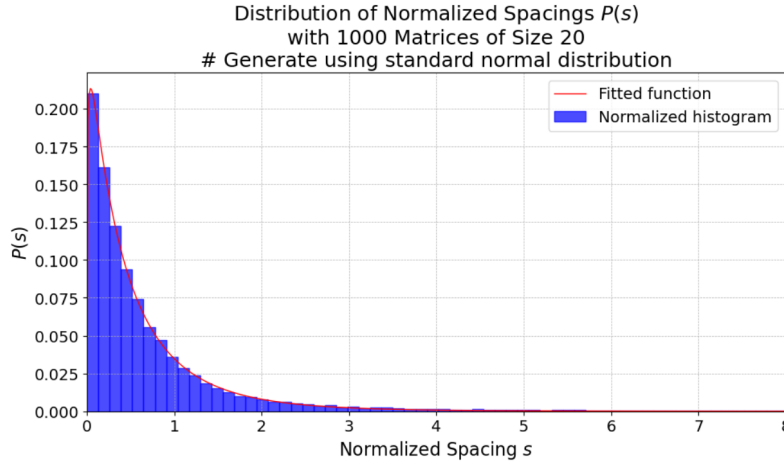
**Figure 19:**  $P(s)$  distribution for hermitian matrices and  $flag=0$



**Figure 20:**  $P(s)$  distribution for hermitian matrices and  $flag=2$

The first thing to notice is once again the presence of the Bell shape of the fit curve, allowing us to perform the same conclusions as before. Additionally, this shows that the quantum random generation actually works fine and provides an alternative method of randomly generating numbers, while future research can be applied to collect larger datasets allowing us to increase the maximum matrix size (now restricted to be  $N \simeq 300$ ).

Afterwards, the diagonal generation of diagonal matrices with real random entries has been tested; in this case, the results are basically all the same for all the flags that have been used for the different generation method. Because of that, I report in Figure 21 and Table 2 the results for  $flag=0$ , while the other parameters stay unchanged:



**Figure 21:**  $P(s)$  distribution for diagonal matrices and  $flag=0$

It is possible no notice in this case the presence of an Poissonian-like distribution: even looking at the best parameters returned by the fit one can see that they are much closer to the situation of  $\alpha = 0$  and  $\beta = 1$ , as suggested from the Theoretical Framework Section. In any case, it is clearly evident the non-repulsive behaviors in eigenvalue spacings, since the peak of the curve is around 0. This highlight the main difference with respect to the previous case of Hermitian matrices.

Finally, the sparse method is applied as a conclusion to this exercise, in order to make comparisons in terms of performance, thus of the CPU execution time, and in terms of numerical results. However, since the generation of diagonal matrices with real random values is basically instantaneous, it make no sense to use the sparse method in this case; on the other hand it is

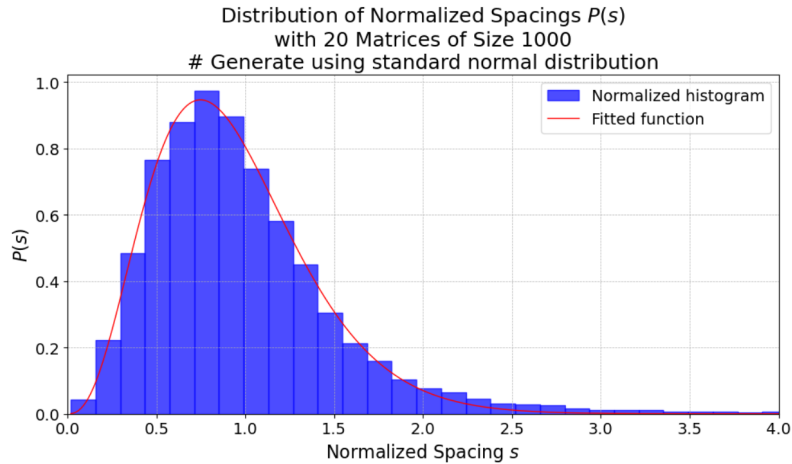


applied to generate hermitian matrices with a fraction of non zero value  $\text{density}=0.1$ . The sparse method is applied using these parameters:

- $N=1000$
- $\text{num\_matrices}=20$
- $N_{\text{bins}}=100$
- $\text{flag}=0$
- $\text{type}=\text{"sparse"}$
- $\text{density}=0.1$

The simulation yields the results summarized in Table 2 and shown in Figure 22. However, considering the CPU time needed to execute the code relative to the sparse method, I noticed that it takes way longer times compared to the classical method of generating hermitian matrices. This is probably due to the fact that the program has to randomly choose indices for the rows and columns to initialize the non-zero values. This consideration makes the sparse method useless in this application, having poorer performances.

On the other end, even though the sparse method is not efficient, one can clearly see from Table 2 that the values of the parameters are similar. The fact that several simulations with different parameters and different methods provide compatible results corroborates the theoretical approach and the correctness of the code in attempting to implement the theory, proving us a very important result from this exercise.



**Figure 22:**  $P(s)$  distribution for sparse matrices and  $\text{flag}=0$

Method	best $a$	best $b$	best $\alpha$	best $\beta$	$\chi^2$
Hermitian (flag 0)	16.72	-3.01	2.74	1.30	$1.6 \cdot 10^{-4}$
Hermitian (flag 2)	10.22	-2.53	2.42	1.39	$8.3 \cdot 10^{-4}$
Diagonal (flag 0)	0.83	-3.17	0.28	0.61	$2.8 \cdot 10^{-5}$
Sparse (flag 0)	14.86	-2.92	2.59	1.29	$6.75 \cdot 10^{-5}$

**Table 2:** Table which summarizes all the parameters best value for the 4 considered simulations

## 5 Conclusion

In this study on random matrix theory, the spectral properties of random matrices were explored through the generation and analysis of eigenvalue spacings for both random Hermitian matrices and real diagonal matrices. The objective was to analyze the normalized spacing distributions  $P(s)$  for different matrix types and to evaluate how these distributions align with theoretical models of eigenvalue statistics.

By fitting these distributions to the function of Equation 6, distinct patterns were observed that reflect the inherent characteristics of each matrix type.

The results revealed a clear distinction between the Hermitian matrices, which exhibited level repulsion typical of the Gaussian Unitary Ensemble (GUE), and the real diagonal matrices, where eigenvalues showed a clustering tendency consistent with Poissonian statistics. The Hermitian matrices' spacing distribution demonstrated a pronounced peak away from zero, indicating that adjacent eigenvalues are unlikely to be very close, as expected from systems with underlying correlations or symmetries. Conversely, the real diagonal matrices displayed a high probability density near zero spacing, consistent with the expected lack of correlation between eigenvalues, a hallmark of independent, unstructured systems.

Additionally, the study underscored the impact of different random number generation methods on the statistical properties of the matrices. The use of quantum random numbers yielded results consistent with classical methods while offering the benefits of intrinsic unpredictability. Moreover, the fitting parameters obtained for  $P(s)$  across different generation flags reaffirmed the robustness of quantum random number generation in capturing complex random matrix properties.

Overall, this investigation into random matrix theory highlighted the effectiveness of spectral analysis for distinguishing matrix types based on eigenvalue distributions and confirmed that the theoretical framework aligns well with empirical findings. The study illustrates that matrix structure—whether Hermitian or real diagonal—significantly influences eigenvalue behavior, providing insights into the statistical mechanics of complex systems and underscoring the relevance of RMT in modeling various phenomena in physics and beyond.

## 6 Appendix

### Quantum Random Number Generation - QRNG

#### 6.1 How to build QRNG bits from Photon Detection

The unpredictability of photon detection events allows us to generate a sequence of truly random numbers. Classical random number generators, such as those based on deterministic algorithms, produce sequences that are fundamentally predictable if the algorithm and initial conditions are known. In contrast, a Quantum Random Number Generator (QRNG) takes advantage of the inherent randomness in quantum mechanics to produce sequences that are unpredictable, even in principle.

In this experiment, random numbers are generated by detecting the arrival times of photons and extracting random bits from these times. The process can be outlined as follows:

1. **Photon Detection:** Each photon detection is recorded with a timestamp, denoted as  $t_i$ , which represents the exact moment the photon was detected.
2. **Time Interval Calculation:** The time intervals  $\Delta t_i = t_{i+1} - t_i$  between successive photon detections are computed. These time intervals follow the exponential distribution described above, reflecting the random nature of photon arrivals.

3. Bit Extraction: For each pair of consecutive time intervals  $\Delta t_i$  and  $\Delta t_{i+1}$ , a bit is generated. If  $\Delta t_{i+1} > \Delta t_i$ , a bit value of 1 is assigned; otherwise, a bit value of 0 is assigned. This method exploits the randomness in the detection times to produce a random sequence of bits.
4. Byte Construction: The generated bits are grouped into blocks of 8 to form bytes. These bytes are then used to create a sequence of random numbers. The statistical properties of the sequence can be analyzed to assess its randomness.

The resulting byte distribution is expected to follow a uniform distribution if the photon detection process is truly random. This uniformity is a key indicator that the QRNG is functioning as expected, producing random data suitable for cryptographic and other applications that require high-quality randomness.

## 6.2 Code and Implementation

Essentially, to consider the implementation of the QRNG, one has to consider the following two functions:

- `bits_to_byte`: it converts a list of 8 individual bits into a single byte by shifting each bit into its proper position within a byte.
- `qrng_list`: it generates a list of bytes from numerical data by first creating a list of bits based on whether each data point is greater than the previous one. These bits are then grouped into chunks of 8 and converted into bytes using `bits_to_byte`, resulting in a list of bytes derived from the original data pattern.

These two functions are used when the QRNG process is called to randomly generate matrices, and, as previously said, it is related to the scenario of `flag=2`. So, let's have a look at the function that generates a diagonal matrix, for instance, using the QRNG (the functions that generate hermitian matrices or sparse matrices work in the exact same way):

```
elif flag == 2:
    path_spin=os.path.abspath('/home/sdruci/Q-OpticsLaser/lab1/Static_wheel/Part_0.txt')
    df0 = pd.read_csv(path_spin, sep=';', header=None, names=['Time_Tag', 'Channel'], skiprows=5)
    path_spin =os.path.abspath('/home/sdruci/Q-OpticsLaser/lab1/Static_wheel/Part_1.txt')
    df1 = pd.read_csv(path_spin, sep=';', header=None, names=['Time_Tag', 'Channel'], skiprows=5)
    path_spin =os.path.abspath('/home/sdruci/Q-OpticsLaser/lab1/Static_wheel/Part_2.txt')
    df2 = pd.read_csv(path_spin, sep=';', header=None, names=['Time_Tag', 'Channel'], skiprows=5)

    df_stat = pd.concat([df0.diff(), df1.diff(), df2.diff()])

    data = np.array(df_stat.Time_Tag)

    bytes_list = qrng_list(data)

    if len(bytes_list) < N:
        raise ValueError("Not enough bytes in bytes_list to fill the matrix.")

    # Randomly sample N*N values from bytes_list
    sampled_bytes = np.random.choice(bytes_list, size=N, replace=False)
    # Normalize to range [-1, 1]
    diagonal_entries = (sampled_bytes / 127.5) - 1 # Scale from [0, 255] to [-1, 1]
else:
    raise ValueError("Invalid flag value. Use 0 for normal distribution or 1 for [-1, 1].")
A = np.diag(diagonal_entries)
return A, diagonal_entries
```

**Figure 23:** Distribution of bytes created from the photon-counts events. An almost uniform distribution is obtained.

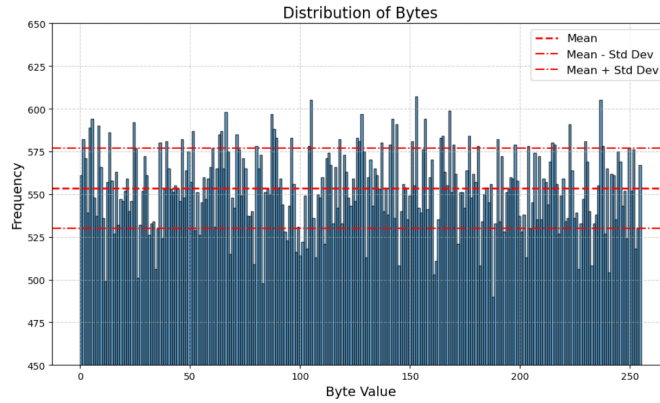
The data are imported and the dataset is created merging the three different sub-datasets, in order to have a better statistics. Then the function `qrng_list` is called in order to create

the list of bytes, and subsequently the function samples  $N$  values from it. As a final step, the sampled bytes are scaled from  $(0, 255)$  to  $(-1, 1)$  range.

### 6.3 Randomness of generated bits

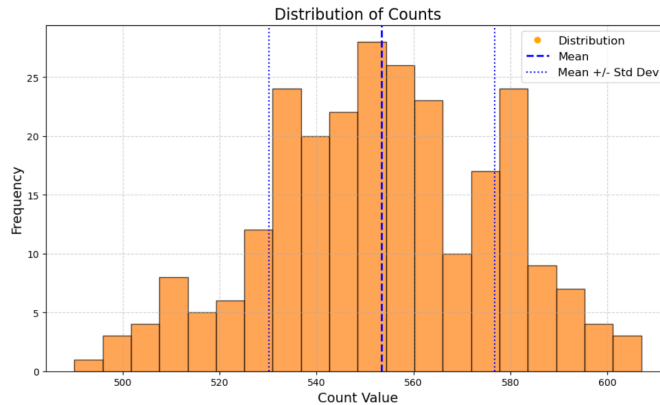
Due to the previous considerations, the photon detection is fundamentally a quantum process. In the context of the Arecchi experiment, so using a Poissonian laser light, the arrival of individual photons is a random quantum event.

Since the events are independent, we can use them to generate a genuine random set of data. The three acquired dataset have been merged together, and a list of random bytes is created in this way: for each pair of data points, the algorithm produces a 1 bit if the current value is greater than the previous one, and 0 otherwise. Combining them in blocks of 8 the string of bits is then converted into bytes. The distribution of bytes is depicted in Figure 24 where it is possible to observe a uniform distribution, almost completely within one standard deviation from the mean value. This clearly indicates that the process is random and suitable for a random extraction.



**Figure 24:** *Distribution of bytes created from the photon-counts events. An almost uniform distribution is obtained.*

Afterwards, a process to verify the presence of periodic artifacts or regular patterns ( a “comb-like” structure) in the data after an extraction is performed. For this reason, the comb distribution about average number of byte appearance is depicted in Figure 25.



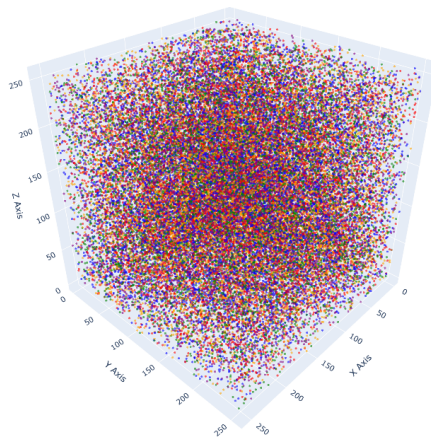
**Figure 25:** *Distribution of the counts of the bytes created from the photon count events*

The “comb” distribution resembles a Bell (normal) distribution, suggesting that the byte

frequencies in the data are centered around a mean value with symmetrical spread or variation on either side.

A Bell-like distribution indicates that most byte appearances are clustered around the average value, with fewer occurrences as one moves away from the mean in either direction. This is typical for natural fluctuations in random processes and suggests that the procedure to build a quantum random number generator is indeed correct.

Moreover, it is possible to provide the distribution of the bytes in three-dimensional space, in order to analyze the presence or lack of clear structure or pattern in the random number data. This is shown in Figure 26, where one can clearly see the absence of such patterns or structures, and since the scatter plot appears uniformly distributed and “foggy,” this suggests that the bytes are behaving randomly.



**Figure 26:** *A 3D representation of the distribution of the bytes, useful to highlight the absence of structures or patterns and the presence of a random data distribution.*

Finally, it is possible to perform other tests in order to quantify the randomness of this Quantum Random Number Generator, compared to the pseudo-random number generator algorithms often used in libraries like `random` from `numpy`.

In particular, the program `ent` is used in order to apply various tests to sequences of bytes. The program is useful for evaluating random number generators for encryption and statistical sampling applications, compression algorithms, and other applications where the information density of a file is of interest.

Among these tests, this program applies:

- The chi-square test: it is a widely method used to assess the randomness of data. It compares the observed byte distribution in a file to the expected distribution for a random sequence. The result is expressed as a percentage, indicating how often a truly random sequence would produce a similar result.
- Arithmetic Mean: Calculates the average byte (or bit) value in a file. For random data, the mean should be around 127.5 for bytes or 0.5 for bits. Deviations from these values indicate consistently high or low values in the data.
- Monte Carlo Value for  $\pi$ : Uses successive 6-byte sequences to estimate the value of  $\pi$  by simulating points inside a square and circle. The accuracy improves with large data sets, and random data should yield a  $\pi$  value close to the real  $\pi$ .
- Serial Correlation Coefficient: Measures the dependency of each byte on the previous one. A value near zero indicates randomness, while values closer to 1 suggest predictable, non-random data. Non-random files like C programs or uncompressed bitmaps tend to have higher correlation coefficients.

Using the bytes produced as previously described and applying this program executing it with `./ent -b data/qrng.bin`, the following output is provided:

```
Entropy = 0.999998 bits per bit.  
Optimum compression would reduce the size of this 1133472 bit file by 0  
percent.  
Chi square distribution for 1133472 samples is 3.91, and randomly would  
exceed this value 4.79 percent of the times.  
Arithmetic mean value of data bits is 0.4991 (0.5 = random).  
Monte Carlo value for Pi is 3.132548488 (error 0.29 percent).  
Serial correlation coefficient is 0.001281 (totally uncorrelated = 0.0).
```

On the other hand, executing the same program with `./ent -b data/prng.bin` on the string of bytes produced using the library `random` from `numpy`, one gets the following results:

```
Entropy = 0.337140 bits per bit.  
Optimum compression would reduce the size of this 9067776 bit file by 66  
percent.  
Chi square distribution for 9067776 samples is 6943734.05, and randomly  
would exceed this value less than 0.01 percent of the times.  
Arithmetic mean value of data bits is 0.0625 (0.5 = random).  
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).  
Serial correlation coefficient is 0.400637 (totally uncorrelated = 0.0).
```

Finally, comparing the two generation techniques with the results of all the tests, it is possible to conclude that the degree of randomness is significantly higher in the quantum version, using the detection of single photons to produce the string of bytes; this underlines the supremacy with the respect to the classical pseudo-random number generation and highlights the potential applications that this method can have in various fields.

## Code

All the data and the code for the data analysis can be found in this public Github repository: <https://github.com/sdracia/QIC.git>