Quantum Information and Computing

Academic Year 2024 - 2025

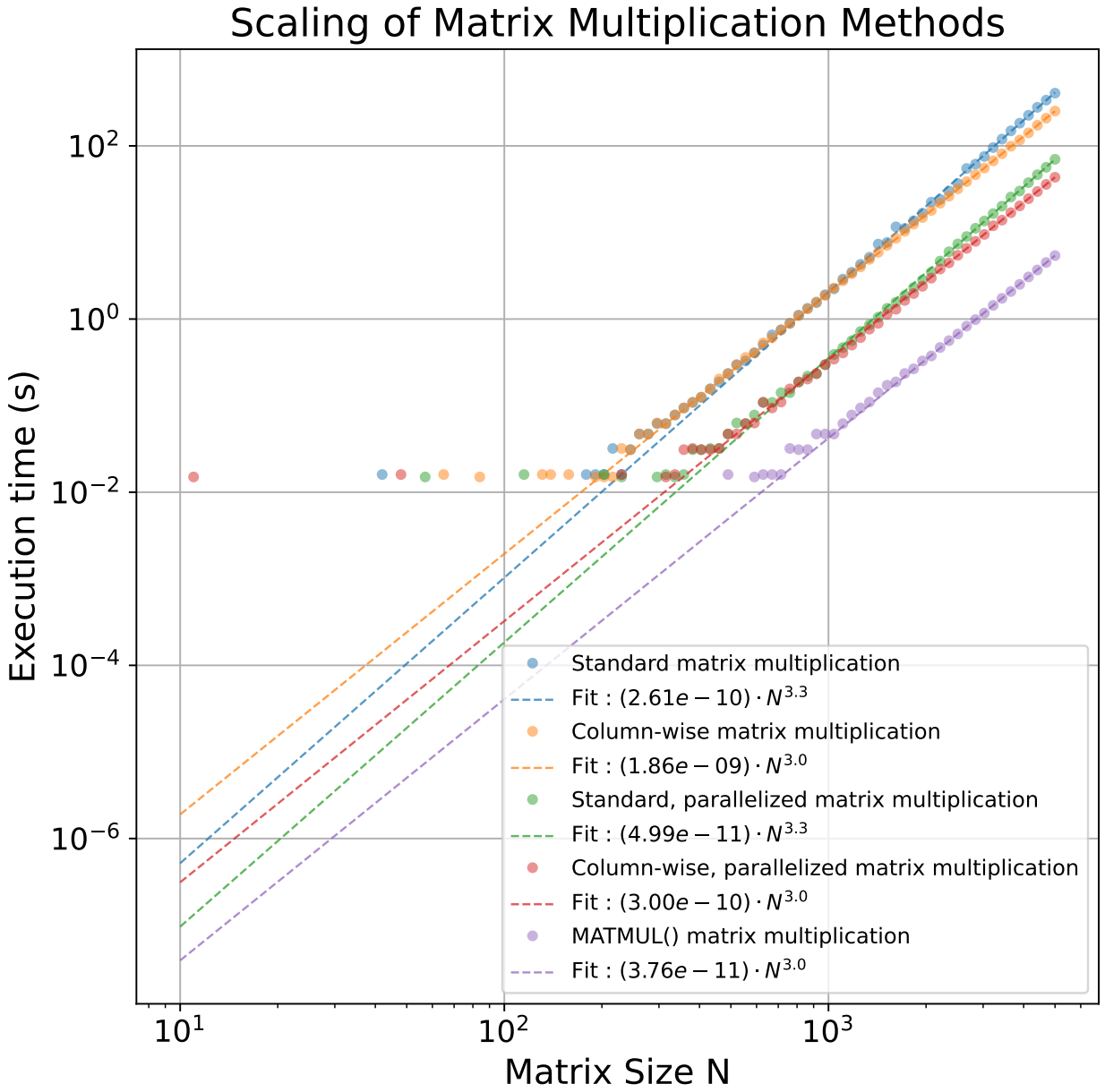
ASSIGNMENT 3

Due by November 11, 2024

Gabriel Amorosetti

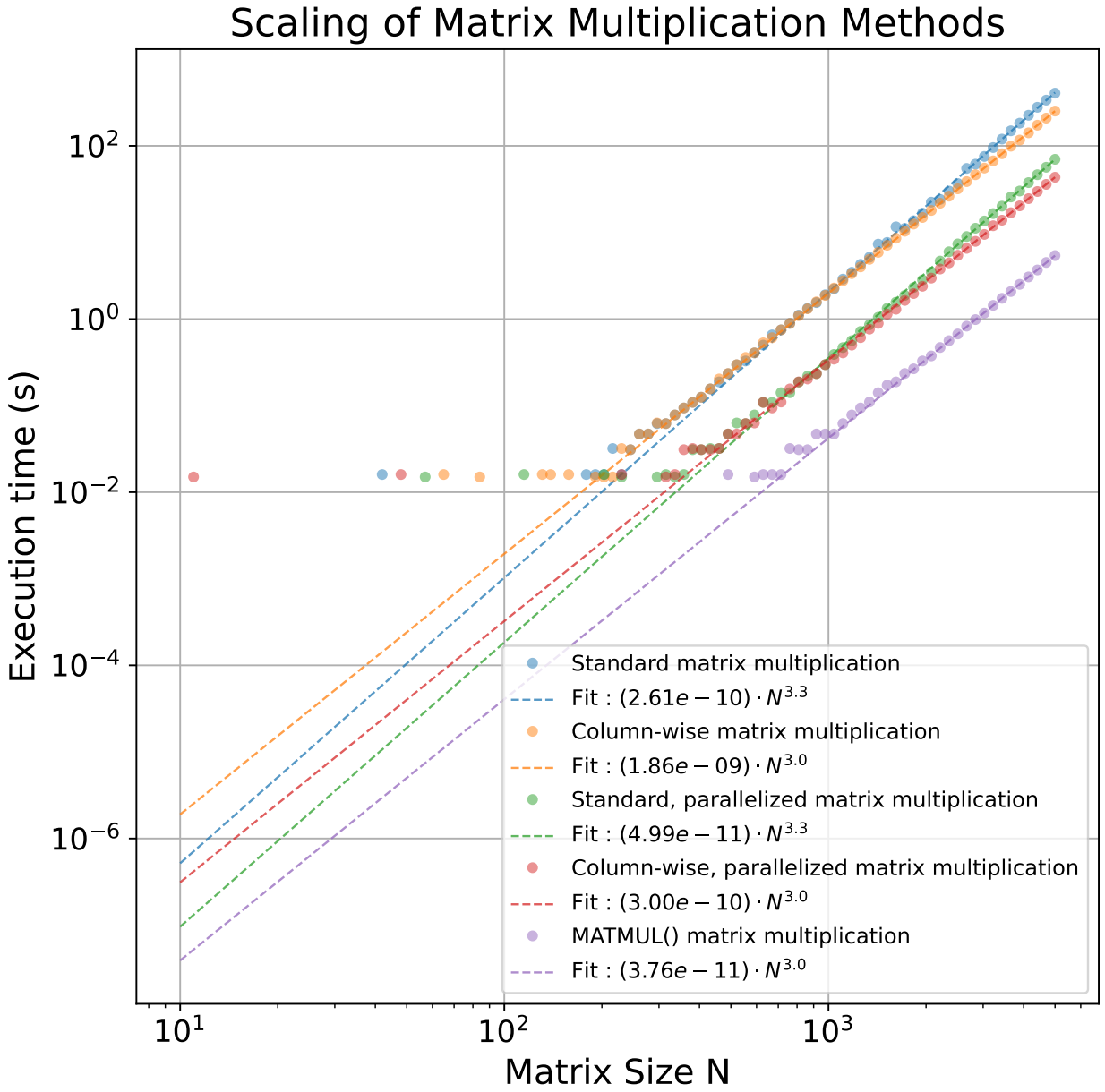
gabriel.amorosetti@studenti.unipd.it

1. Scaling of the matrix-matrix multiplication



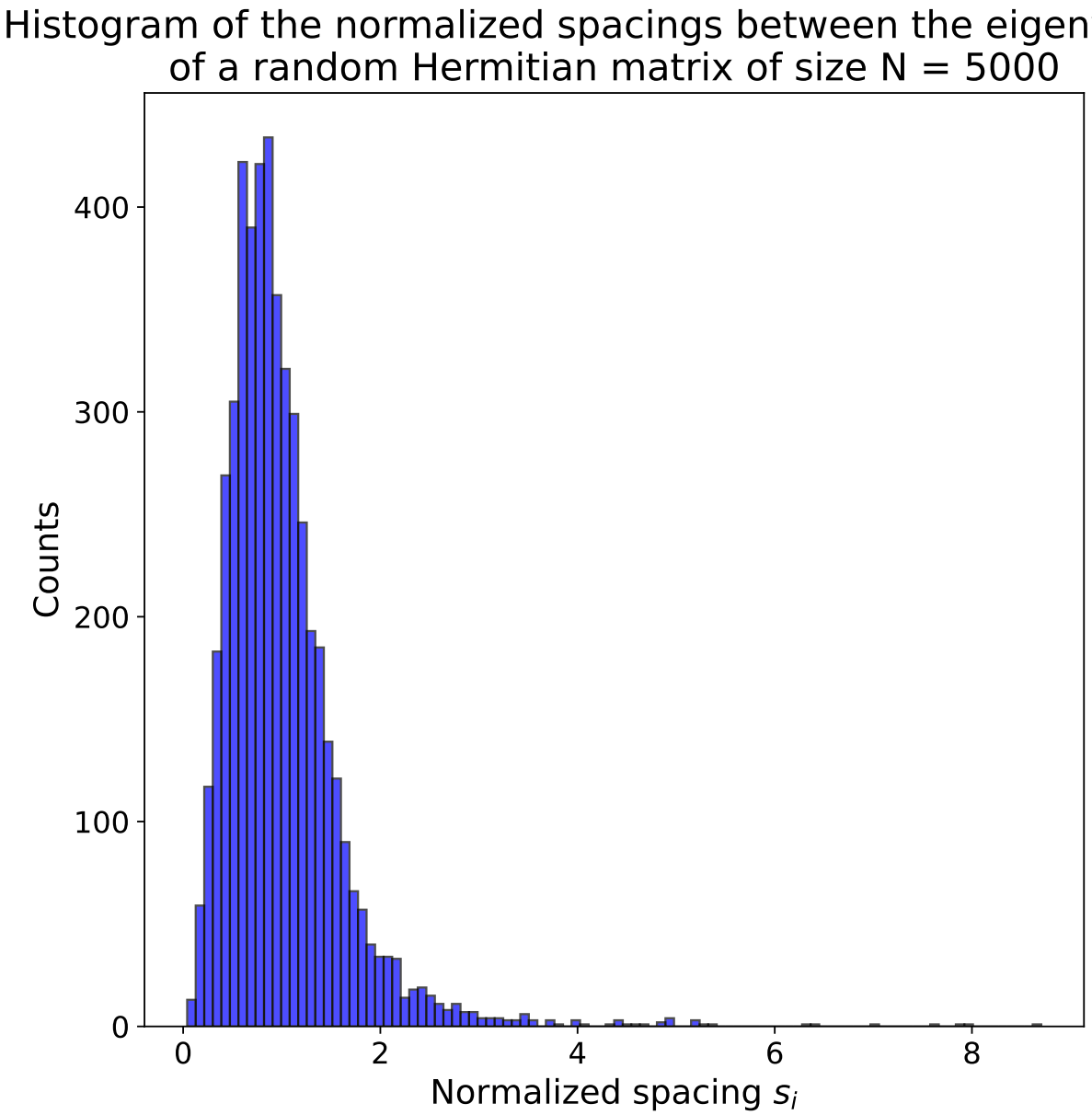
- Creation of Python script that compiles the matrix-matrix multiplication Fortran code, and execute it for 100 different matrix sizes, logarithmically spaced between 10 and 5000
- For each size, the multiplication is computed with all the implemented methods, and the execution times are stored on a file for each method
- The execution times are fitted depending on the input size N for each method to the function $a \cdot N^b$
  - Performed with the `curve_fit` function of the `scipy.optimize` package
  - Fits represented linearly by ***log(a) + b·log(n)*** on a log-scale plot

1. Scaling of the matrix-matrix multiplication



- Except for smaller sizes, all the multiplication methods show a polynomial scaling, as the computation times grow linearly on a log-scale
  - For small sizes, the computation may be faster than the time to access the data needed, so the execution time can't be lower than a certain threshold
- As expected, all the methods scale to n^3 or close to, as the matrix multiplication requires n operations of n^2 elements
- The standard implementations of the matrix multiplication are the least efficient and the column-wise implementations scale better and closer to n^3, because of the major-column order
  - The parallelized versions of these two methods have a lower intercept, showing an improvement of the computation times
- The built-in Fortran function, having the lowest intercept a and scaling to n^3, is the more efficient

2. Eigenproblem

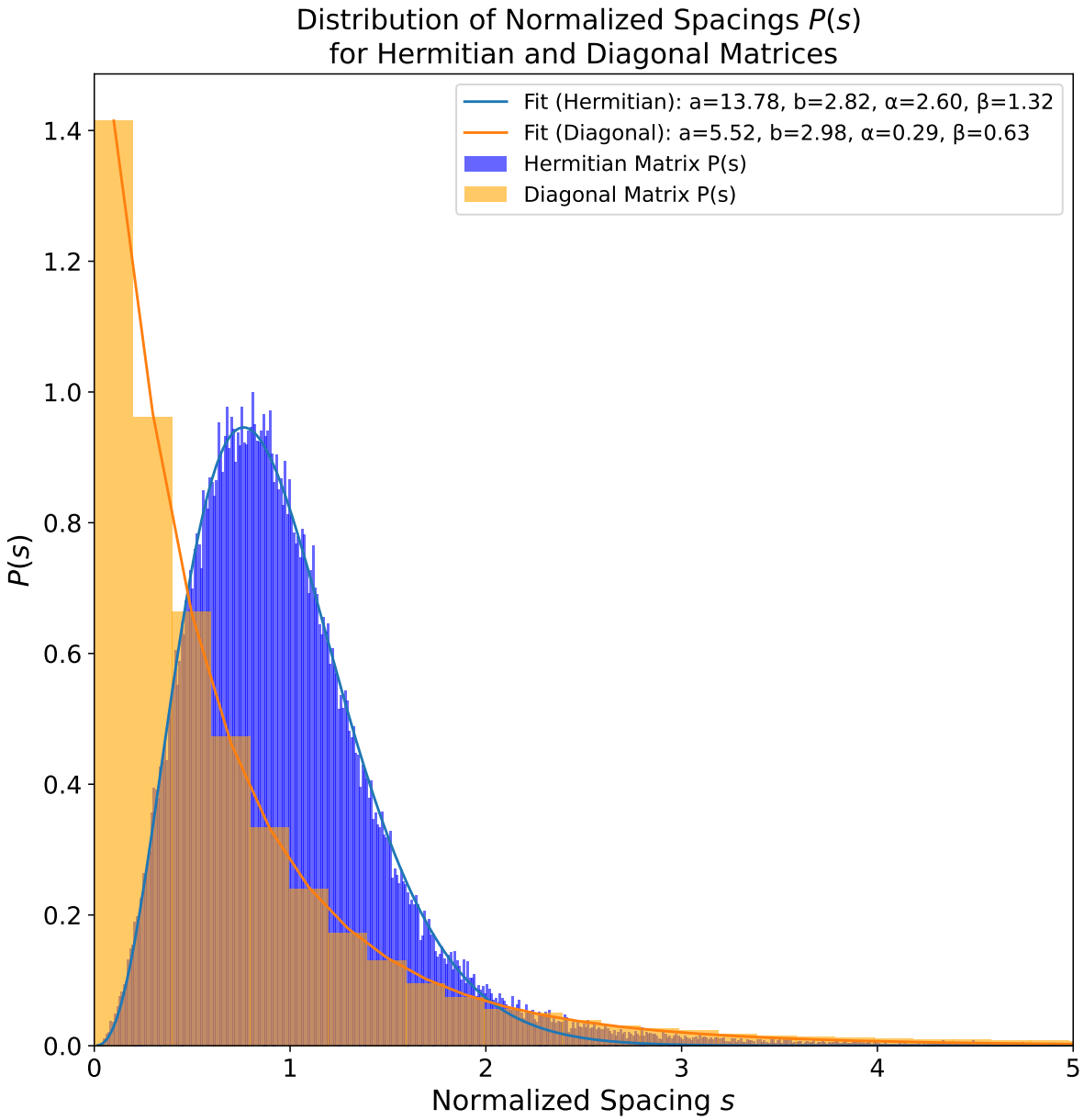


- Generation of a random complex Hermitian matrix A of size N = 5000
  - Using symmetries and only storing the lower triangle of the matrix, we reduce the number of operations : $n^2 \rightarrow n(n+1)/2$
- Diagonalization of A and storing its eigenvalues λ_i in ascending order, with the built-in numpy functions
- Computing the normalized spacings :

  1) Spacings : $\Lambda_i = \lambda_{i+1} - \lambda_i$
  2) Normalized spacings : $S_i = \Lambda_i / average(\Lambda_i)$

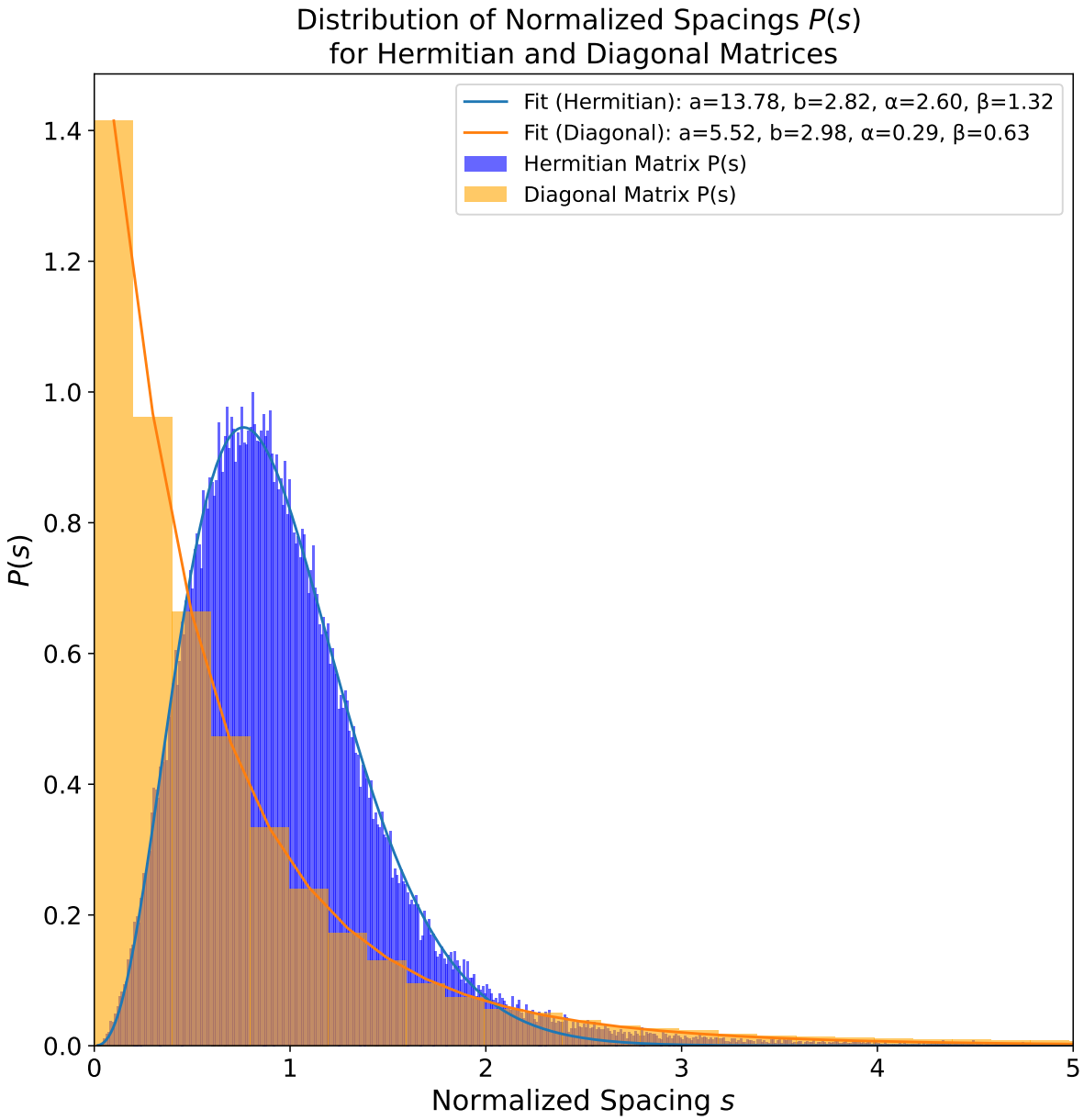
- Representing the statistical distribution of the normalized spacings with an histogram

3. Random matrix theory



- Generation of 100
  - Random complex Hermitian matrices of size N = 1000, using the method implemented previously
  - Diagonal matrices of size N = 1000
- Diagonalization of the matrices and storing their eigenvalues λ_i in ascending order, with the built-in numpy functions
- Computing the normalized spacings S_i using the method implemented previously and averaging the results from the 100 matrices
- Representing the distribution of the normalized spacings S_i for each matrix type
- Fitting each distribution to the Wigner-Dyson distribution $W(s) = a \cdot s^{\alpha} \cdot exp(b \cdot s^{\beta})$

3. Random matrix theory



- Each distribution is fitted to the Wigner-Dyson distribution $W(s) = a \cdot s^{\alpha} \cdot exp(b \cdot s^{\beta})$
  - For random complex Hermitian matrices, the behaviour is well described by the Wigner-Dyson model
  - For real random diagonal matrices however, α and β are close to 0, modelling instead an exponential law $a \cdot exp(b)$