Quantum Information and Computing
Assignement 4 Report

Andrea Turci

November 2024

Quantum Harmonic Oscillator

1 Introduction

The quantum harmonic oscillator is one of the most fundamental and extensively studied systems in quantum mechanics. It represents a particle subjected to a restoring force proportional to its displacement, encapsulated in the potential $V(x) = \frac{1}{2}\omega^2 x^2$. Near equilibrium points, many potential energy functions can be approximated by the harmonic oscillator, making its study relevant across various physical contexts. Furthermore, its well-defined mathematical framework provides an excellent benchmark for testing numerical methods and validating computational approaches.
This report focuses on solving the one-dimensional quantum harmonic oscillator using both analytical and numerical techniques. The analytical solution is derived through the properties of Hermite polynomials, yielding exact expressions for the eigenvalues and eigenfunctions.
On the numerical side, the finite difference method (FDM) is employed to discretize the Hamiltonian operator and transform the continuous eigenvalue problem into a matrix eigenvalue problem. This approach involves constructing the Hamiltonian matrix as the sum of a tridiagonal or pentadiagonal kinetic energy matrix and a diagonal potential energy matrix. Numerical solutions are validated by comparing them to analytical results, focusing on correctness, stability, accuracy, and efficiency.
In addition to the FDM, the report explores the use of sparse matrices to optimize computations for large matrix sizes. Sparse matrices, characterized by their predominantly zero-valued entries, offer significant advantages in terms of memory and computational time when handling large-scale problems.

2 Theoretical Framework

2.1 The Quantum Harmonic Oscillator

The quantum harmonic oscillator is a fundamental system in quantum mechanics, representing a particle subjected to a restoring force proportional to its displacement. This system serves as a cornerstone for understanding more complex quantum phenomena, as many potential energy functions can be approximated as harmonic oscillators near equilibrium points. Furthermore, its exact solvability provides an excellent benchmark for testing numerical methods.

The one-dimensional time-independent Schrödinger equation for a quantum harmonic oscillator is given by:

$$\hat{H}\Psi(x) = E\Psi(x),$$

where $\hat{H}$ is the Hamiltonian operator:

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}m\omega^2 x^2.$$

Here, the first term corresponds to the kinetic energy operator, and the second term represents the harmonic potential energy. The parameters m and ω denote the particle mass and angular frequency, respectively, while $\Psi(x)$ is the wavefunction, and E is the associated energy eigenvalue.

Using natural units where $\hbar = m = 1$, the Hamiltonian simplifies to:

$$\hat{H} = -\frac{1}{2}\frac{d^2}{dx^2} + \frac{1}{2}\omega^2 x^2.$$

The goal is to determine the eigenvalues E_n and eigenfunctions $\Psi_n(x)$ that solve this eigenvalue problem. These quantities provide the quantized energy levels and the corresponding spatial probability distributions for the system.

The analytical solution to the quantum harmonic oscillator leverages the properties of Hermite polynomials. The eigenfunctions are given by:

$$\Psi_n(x) = \sqrt{\frac{1}{2^n n!}}\left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right),$$

where $H_n(x)$ are the Hermite polynomials of degree n, and $n = 0, 1, 2, \ldots$ is the quantum number. These wavefunctions exhibit alternating symmetry: even n results in even functions, while odd n results in odd functions.

The corresponding eigenvalues are:

$$E_n = \hbar\omega\left(n + \frac{1}{2}\right).$$

In natural units ($\hbar = 1$), the eigenvalues simplify to:

$$E_n = \omega\left(n + \frac{1}{2}\right).$$

These solutions illustrate the quantized nature of energy levels, a defining characteristic of quantum systems. The wavefunctions describe the probability density of finding the particle at a given position, with the ground state $n = 0$ being Gaussian-shaped, while higher states exhibit increasing oscillatory behavior.

2.2 Finite Difference Method (FDM)

The finite difference method (FDM) is a numerical technique that approximates derivatives by finite differences, transforming differential equations into algebraic equations. This method is well-suited for solving the Schrödinger equation numerically, especially when analytical solutions are unavailable or cumbersome.

To apply the FDM, the spatial domain $[a, b]$ is divided into N intervals, each of width:

$$\Delta x = \frac{b-a}{N}.$$

The wavefunction $\Psi(x)$ is represented at discrete grid points $x_i = a + i\Delta x$, for $i = 0, 1, \ldots, N$. Substituting these discretized points into the Schrödinger equation transforms it into:

$$\hat{H}\Psi = E\Psi,$$

where Ψ is now a vector, and $\hat{H}$ is a matrix representing the Hamiltonian operator.

The Hamiltonian matrix is composed of two parts: the kinetic energy matrix $\hat{K}$ and the potential energy matrix $\hat{V}$:

$$\hat{H} = \hat{K} + \hat{V}.$$

Kinetic Energy Matrix ($\hat{K}$) The kinetic energy operator involves the second derivative, which is approximated using finite differences. For a second-order scheme, the second derivative is:

$$\frac{d^2\Psi}{dx^2} \approx \frac{\Psi_{i+1} - 2\Psi_i + \Psi_{i-1}}{\Delta x^2}.$$

This yields a tridiagonal kinetic energy matrix:

$$\hat{K} = \frac{1}{2\Delta x^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}.$$

For a higher-order approximation (e.g., fourth-order), additional terms are included:

$$\frac{d^2\Psi}{dx^2} \approx \frac{-\Psi_{i+2} + 16\Psi_{i+1} - 30\Psi_i + 16\Psi_{i-1} - \Psi_{i-2}}{12\Delta x^2}.$$

This results in a pentadiagonal kinetic energy matrix.

Potential Energy Matrix ($\hat{V}$) The potential energy matrix is diagonal, with each diagonal element corresponding to the potential energy at a grid point:

$$\hat{V}_{ii} = \frac{1}{2}\omega^2 x_i^2.$$

2.2.1 Numerical Solution

Once the Hamiltonian matrix $\hat{H} = \hat{K} + \hat{V}$ is constructed, the next step is to solve the eigenvalue problem. The goal is to find the eigenvalues E_n and eigenvectors $\Psi_n(x)$ of the Hamiltonian matrix, which correspond to the quantized energy levels and the spatial probability distributions of the particle, respectively.

To solve the eigenvalue problem numerically, we perform the diagonalization of the Hamiltonian matrix. The eigenvalues obtained from the diagonalization correspond to the quantized energy levels of the system, and the eigenvectors provide the discretized wavefunctions.

One important consideration in this process is the choice of boundary conditions, which must be imposed on the grid points. For the quantum harmonic oscillator, typical boundary conditions are:

$$\Psi(a) = \Psi(b) = 0,$$

where a and b are the spatial limits of the grid. These boundary conditions ensure that the wavefunction vanishes at the edges, which is consistent with the behavior of the quantum harmonic oscillator in an infinite potential well.

The finite difference method's effectiveness in solving the Schrödinger equation depends on several key properties: accuracy, stability, efficiency, and correctness.

- **Accuracy**: The accuracy of the FDM increases as the grid spacing Δx is reduced, allowing for a better approximation of the continuous differential operators. A finer grid results in a more accurate representation of the Hamiltonian matrix and leads to a closer match between the numerical eigenvalues and the exact analytical values. Additionally, higher-order finite difference schemes, such as the fourth-order scheme, provide better approximations for the derivatives compared to the standard second-order method, but at a higher computational cost.

- **Stability**: Stability is crucial for obtaining reliable numerical results. Stability in the context of the FDM means that the solution does not exhibit unphysical oscillations or blow up due to numerical errors. This is particularly important when dealing with higher-order discretization schemes or larger grid sizes. Stability can be ensured by choosing appropriate boundary conditions and using a sufficiently fine grid. Additionally, the structure of the Hamiltonian matrix, typically tridiagonal or pentadiagonal, ensures that the numerical diagonalization process remains stable and efficient.

- **Efficiency**: The FDM benefits from the efficient matrix structure of $\hat{H}$, particularly the tridiagonal or pentadiagonal form of the kinetic energy matrix $\hat{K}$. This structure allows for fast numerical diagonalization, which is critical for large-scale problems. The matrix can be diagonalized using specialized algorithms that take advantage of its sparse structure, leading to a significant reduction in computational time compared to methods that require full matrix inversion. However, the efficiency can still be impacted by the number of grid points and the complexity of the potential energy function.

- **Correctness**: Correctness refers to the degree to which the numerical method produces results that are consistent with the true physical solutions. In the case of the quantum harmonic oscillator, correctness is ensured by validating the numerical solutions against the known analytical results. The eigenvalues obtained from the numerical diagonalization should match the analytical energy levels $E_n = \omega(n + \frac{1}{2})$ for each quantum number n. Similarly, the eigenvectors, which represent the wavefunctions, should exhibit the expected behavior, including the correct symmetry (even or odd) for each n.

- **Flexibility**: The finite difference method is highly flexible and can be easily adapted to different types of quantum systems. It can handle a wide range of potential functions, from simple harmonic potentials to more complex, non-analytic potentials. The method can also be extended to higher-dimensional problems, such as the two- or three-dimensional quantum harmonic oscillator, by appropriately discretizing the spatial grid in multiple dimensions.

To validate the numerical solutions, we compare the eigenvalues and eigenvectors obtained from the FDM with the analytical solutions. This comparison allows us to quantify the errors and assess the performance of the method. The key metrics for validation are:

- **Relative error in eigenvalues**: The relative error between the numerical eigenvalues E_n^{num} and the analytical eigenvalues E_n^{ana} is given by:

$$\text{Relative Error} = \frac{|E_n^{\text{num}} - E_n^{\text{ana}}|}{|E_n^{\text{ana}}|}.$$

  This error should be small, and its magnitude decreases as the grid spacing Δx is reduced. A small relative error indicates that the numerical method is providing accurate results.

- **Dot product similarity for eigenvectors**: The similarity between the numerical and analytical wavefunctions can be quantified using the dot product of the corresponding eigenvectors:

$$\text{Error} = 1 - |\langle \Psi_n^{\text{num}}, \Psi_n^{\text{ana}} \rangle|.$$

This measure indicates how well the numerical wavefunction matches the analytical wavefunction. For a perfect match, the error will be zero, and as the grid resolution improves, the error tends to decrease. The dot product is particularly useful for ensuring that the numerical wavefunction exhibits the correct symmetry (even or odd) for each energy level.

In addition to these standard metrics, further validation can include checking the normalization of the wavefunctions and ensuring that the numerical results converge as the grid spacing Δx is refined. Convergence tests can help to identify the optimal grid size that balances accuracy and computational efficiency.

2.3 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

The assignment has been developed using Python as language, due to it's flexibility for operations with matrices and vectors, performing the diagonalizations and solving the eigenvalue problems. In particular, several functions have been defined and split across different files according to their functionalities. For this reason, 4 files have been created:

- `analytical_solution.py`
- `functions.py`
- `sparse_matrix.py`
- `debugger.py`

I will now perform a description for each of these files focusing on the functions they contain that are the most relevant in the discussion (in particular, I will neglect the functions related to the plot of the results).
To better understand how these functions work, I first have to define which are the relevant parameters in our problem, which are going to be changed according to the different situations. These parameters influence the discretization, physical properties, and computational accuracy of the system:

- `N`: This parameter represents the number of grid points used to discretize the spatial domain. A larger `N` results in a finer grid and higher accuracy but increases computational cost. The grid spacing, `deltax`, is derived as:

$$\texttt{deltax} = \frac{2\texttt{L}}{\texttt{N}},$$

ensuring uniform spacing over the domain $[-\mathtt{L}, \mathtt{L}]$. The grid points themselves are stored in `x_i`.

- `omega`: This is the angular frequency of the quantum harmonic oscillator. It determines the curvature of the potential $V(x) = \frac{1}{2}\omega^2 x^2$, and consequently the spacing of the energy levels. Larger values of `omega` correspond to a steeper potential and more tightly spaced wavefunctions.
- `L`: This parameter defines the half-width of the spatial domain $[-\mathtt{L}, \mathtt{L}]$. Increasing `L` enlarges the physical region covered by the grid, accommodating more extended wavefunctions but requiring finer grid resolution to maintain accuracy.
- `k`: This is the number of eigenvalues and eigenvectors considered in the analysis. It specifies how many energy levels and wavefunctions are computed, compared, and plotted. Larger `k` allows for the study of higher excited states but increases computational effort.
- `order`: This parameter specifies the accuracy order of the finite difference scheme used to approximate the kinetic energy operator. The supported values are:
  - `2`: Second-order accuracy, which uses a tridiagonal matrix for the kinetic energy.
  - `4`: Fourth-order accuracy, which employs a pentadiagonal matrix for higher precision at the cost of additional computation.

The parameters `deltax` and `x_i` depend on `N` and `L`. Specifically, `deltax` determines the spacing between grid points, while `x_i` is an array of positions calculated as:

$$\mathtt{x\_i}[i] = \left(\frac{2\mathtt{L}}{\mathtt{N}} \cdot i\right) - \mathtt{L}, \quad i = 0, 1, \ldots, \mathtt{N} - 1.$$

These derived parameters ensure the grid adapts dynamically to changes in `N` and `L`, preserving the uniformity and accuracy of the spatial representation.

Now let's describe the content of each Pythonic file

3.1 analytical_solution.py

The file `analytical_solution.py` contains several functions designed to analytically solve the quantum harmonic oscillator problem, leveraging the mathematical properties of Hermite polynomials and other well-known quantum mechanics formulas. Below, the main functions are described in detail.

1. `hermite(x, n)` This function computes the Hermite polynomial of order `n` over a real-space grid `x`. The implementation begins by ensuring that the input order `n` is non-negative, as negative values are neither physically meaningful nor part of the standard mathematical definition. To perform the computation, the function creates an array of coefficients, all initially set to zero except for the one corresponding to the order `n`, which is set to 1. Then, using `numpy`, the function `np.polynomial.hermite.hermval` evaluates the polynomial at every point in the grid `x`. The result is a Hermite polynomial of the desired order evaluated over the input domain.

2. `harmonic_en(omega=1.0, n=0)` This function calculates the energy levels of a quantum harmonic oscillator. Using natural units, this function follows the formula $E_n = \omega\left(n + \frac{1}{2}\right)$. Also, it ensures that the quantum number `n` is non-negative before proceeding, and then computes and returns the energy associated with the specified level.

3. harmonic_wfc(x, omega, n) This function generates the normalized wavefunction of the quantum harmonic oscillator for a given quantum number `n`, angular frequency `omega`, and spatial grid `x`. The wavefunction is defined by a combination of three components: a normalization prefactor, an exponential decay term $e^{-\frac{m\omega x^2}{2\hbar}}$, and the Hermite polynomial $H_n(x)$. Before performing the computation, the function checks that `n` is non-negative and assumes `omega` is strictly positive. The result is a wavefunction array that represents the probability amplitude for the quantum state `n` at each point in the grid.

4. analytic_eigenv(x_i, omega, k) This function computes the first `k` eigenvalues and eigenvectors of the quantum harmonic oscillator, evaluated over a real-space grid `x_i`. It works by iteratively calculating the energy eigenvalues and wavefunctions for the first `k` quantum states. For each state, the function computes the wavefunction using `harmonic_wfc` and normalizes it by dividing by its norm., while the energy eigenvalues are calculated using the `harmonic_en` function. It then returns an array containing the eigenvalues and another array containing the normalized eigenvectors.

3.2 functions.py

1. kinetic_gen(size, deltax, order=2) This function is designed to generate the kinetic energy matrix for a discretized quantum system, also allowing flexibility in the order of accuracy for the finite difference approximation of the second derivative.
The parameter `size` specifies the dimensions of the square matrix to be created, while `deltax` defines the spacing between grid points in the discretized domain. The optional parameter `order` determines the order of accuracy: `order=2` using a tridiagonal matrix or `order=4`, using a pentadiagonal matrix. Depending on the specified order, the matrix is constructed with appropriate diagonals using `numpy`'s `diag` function. If an unsupported order is provided, the function prints an error message and terminates without returning a result.

2. potential_gen(size, x_i, omega) This function constructs the potential energy matrix for a quantum harmonic oscillator, where the angular frequency of the harmonic oscillator is provided by the parameter `omega`.

Internally, the function first computes a scaling factor $\frac{\omega^2}{2}$ and then multiplies it by the squared position values x_i^2 to generate the main diagonal of the matrix. Since the potential energy depends only on the position, the resulting matrix is diagonal.

3. hamiltonian_gen(size, deltax, x_i, omega, order=2) This function generates the Hamiltonian matrix for a discretized quantum harmonic oscillator and computes its eigenvalues and eigenvectors. To begin, the function invokes `kinetic_gen` to generate the kinetic energy matrix K, and also calls `potential_gen` to construct the potential energy matrix V. The Hamiltonian matrix H is then computed as the sum of K and V.
Next, the function diagonalizes the Hamiltonian using `numpy.linalg.eigh`, which computes its eigenvalues and eigenvectors which are sorted in ascending order. Additionally, the eigenvectors are normalized to have a unit norm, and their signs are adjusted for consistency. The adjustments include ensuring symmetry about the central grid point and enforcing positivity at specific reference points, depending on the state parity.

4. correctness(k, eigval, eigvec, eigval_analy, eigvec_analy) This function evaluates the accuracy of numerical eigenvalues and eigenvectors by comparing them with analytical solutions. This comparison is essential for validating the numerical approach and quantifying its precision.

The function focuses on two aspects:

1. Eigenvalue Errors: It computes the relative errors between the numerical and analytical eigenvalues. The relative error for the n-th eigenvalue is defined using the previous Formula, using in this way a metric able tp measures the deviation of the numerical results from the exact solutions.

2. Eigenvector Similarity: It calculates the difference between numerical and analytical eigenvectors using the dot product. Before comparison, both eigenvectors are normalized to ensure consistency. The similarity is quantified using the previous Formula, in such a way that smaller values indicate a closer match between the numerical and analytical wavefunctions.

5. `stability(num_runs, order, k, N, deltax, x_i, omega)` This function analyzes the stability of numerical solutions for the quantum harmonic oscillator by evaluating the variability of eigenvalues and eigenvectors over multiple runs. This is one of the crucial aspects the assignment focuses on, since we want to ensure that repeated calculations yield consistent results despite potential variations in computation.
The function generates the Hamiltonian and computes its eigenvalues and eigenvectors `num_runs` times for the given discretization order (`order`), grid size (`N`), grid spacing (`deltax`), and angular frequency (`omega`).
Then, it performs the statistical analysis of eigenvalues: for each energy level up to `k`, the function calculates the mean and standard deviation of the eigenvalues across the runs. A smaller standard deviation indicates greater stability.
Finally, in order to assess the consistency of eigenvectors the function computes the dot product differences between eigenvectors from consecutive runs. These differences measure variability in wavefunction alignment and are aggregated into a matrix for analysis.

6. `discretization_size(N_min, N_max, step, k, omega, L, order)` This function investigates the effect of grid size on the accuracy of numerical solutions for the quantum harmonic oscillator. By varying the number of grid points, it quantifies how discretization impacts the precision of eigenvalues and eigenvectors. First, this function iterates over grid sizes from `N_min` to `N_max` in steps of `step`. For each size, the spatial grid is defined over the domain $[-L, L]$, with grid spacing Δx adjusted accordingly.
Then, for each grid size, the function constructs the Hamiltonian matrix, computes its eigenvalues and eigenvectors, and compares them with analytical solutions. The number of energy levels and wavefunctions analyzed is determined by `k`.
As a final thing, it calculates the relative errors of numerical eigenvalues with respect to analytical ones, and the dot product differences between numerical and analytical eigenvectors.

7. `omega_variation(omega_min, omega_max, omega_step, k, N, L, order)` This function examines how the angular frequency ω affects the accuracy of numerical solutions for the quantum harmonic oscillator. This analysis is important because ω determines the curvature of the potential well and influences the spacing of energy levels.
Differently from the previous function, instead this function iterates over values of ω from `omega_min` to `omega_max` in steps of `omega_step`, and for each value of ω, the function generates the Hamiltonian matrix based on a fixed grid size (`N`) and spatial domain $[-L, L]$. After computing the eigenvalues and eigenvectors and comparing with the analytical solutions, it again perform the error calculation for the eigenvalues and eigenvectors in the same way as before.

3.3 `sparse_matrix.py`

1. `sparse_matrix_one_eig(L, omega, sizes, order)` This function evaluates and compares the computational performance of sparse and dense matrix methods when computing

the lowest eigenvalue of the Hamiltonian.
For each matrix size in the list `sizes`, the function constructs the Hamiltonian matrix $H = K + V$, and convert it into a sparse format using `scipy.sparse.csc_matrix`. Two methods are then used to compute the smallest eigenvalue: a sparse solver (`scipy.sparse.linalg.eigsh`), and a dense solver (`numpy.linalg.eigh`).
For each solver, the computational time is recorded, and then plotted as a function of the matrix size. The x-axis uses a logarithmic scale to better represent the wide range of sizes.

2. `sparse_matrix_heatmap(L, omega, sizes, order, num_eig)` This function generates a heatmap that compares the computational time difference between sparse and dense methods for calculating multiple eigenvalues of the Hamiltonian.
In particular, the function repeats the same process the previous one uses for the generation of a sparse matrix describing the Hamiltonian of our system. Then, for each value of eigenvalues to compute (from 1 to `num_eig`, it uses a dense solver and a sparse solver to compute the specified number of smallest eigenvalues using the 2 different methods. The difference between dense and sparse computation times (`dense_timing - sparse_timing`) is calculated and stored in a `timings` array.
Finally, a heatmap is generated using `seaborn`, where the rows correspond to the number of eigenvalues computed (`num_eig`), and columns represent different matrix sizes (`sizes`). Each cell shows the time difference between dense and sparse computations for the corresponding parameters.

3.4 `debugger.py`

1. `checkpoint(debug, verbosity=1, msg=None, var1=None, var2=None, var3=None)`
This function serves as a versatile debugging tool that provides detailed feedback about the state of the program during execution.
The parameter `debug` acts as a master switch to enable or disable debugging output. If `debug` is set to `False`, the function exits immediately without producing any output.
Then, depending on the verbosity level (it can be equal to 1,2 or 3), the function displays a message with a level of detailed specified by the verbosity value.
Finally, the optional `msg` parameter allows users to provide context-specific messages, while the variables (`var1`, `var2`, `var3`) can be displayed with custom labels for clarity. The function leverages the helper function `print_variable` to format and print the variables with their labels.

4 Results

Let's start to analyze the results that have been obtained.
In particular we start analyzing the Analytical results yielded by the functions defined in `analytical_solution.py`. The first k eigenvectors and eigenvalues of the harmonic oscillator problem have been generated, choosing a grid in the spatial domain $[-L, L]$, with `L=5`, `N=1000` points for the discretization, `omega=1` and `k=5`. The result are shown in Figure 1 and 2, where one can see the amplitudes of the first 5 analytical eigenvectors as a function of the position, and the correspondent analytical eigenvalues.

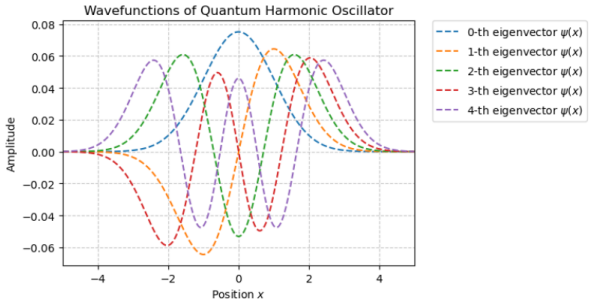

Figure 1: *Amplitudes of first 5 analytical eigenvectors*

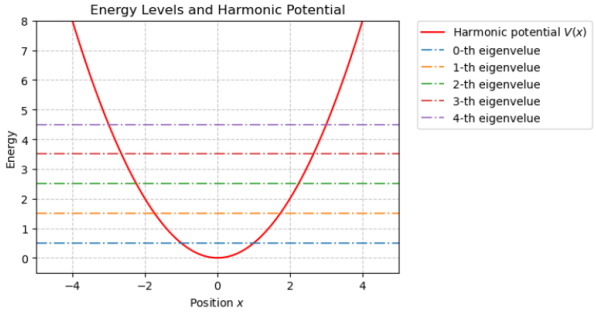

Figure 2: *Values for the first 5 analytical eigenvalues*

Our first goal is to perform a comparison of these first results with the numerical values obtained from the simulations. For this reason, the eigenvectors and eigenvalues are numerically computed using the function `hamiltonian_gen`, keeping the same input parameters used before, for both the orders in the expansion (`order=2,4`). First, in order to see the correctness of our generation, a visual depiction has been produced, similarly as before.
For `order=2` the results are the following:

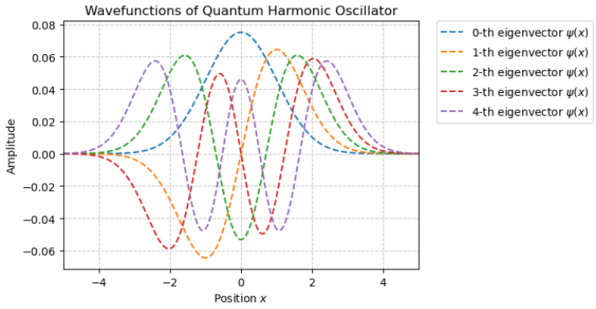

Figure 3: *Amplitudes of first 5 eigenvectors with `order=2`*

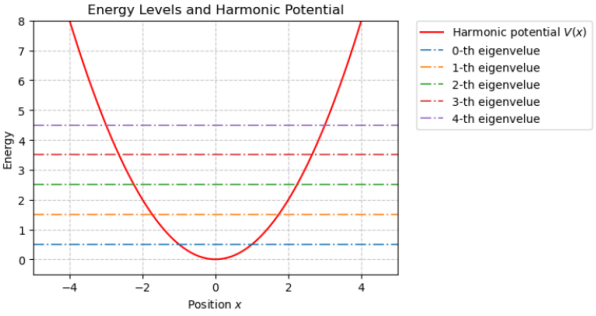

Figure 4: *Values for the first 5 eigenvalues with `order=2`*

while for `order=4` the results are the following:

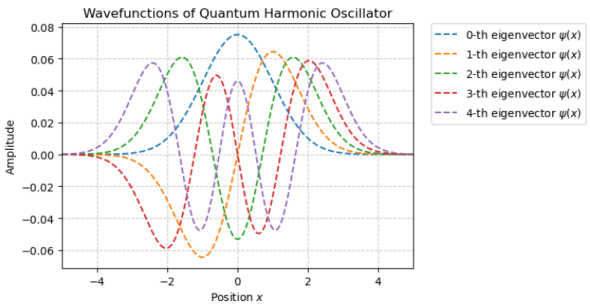

Figure 5: *Amplitudes of first 5 eigenvectors with `order=4`*

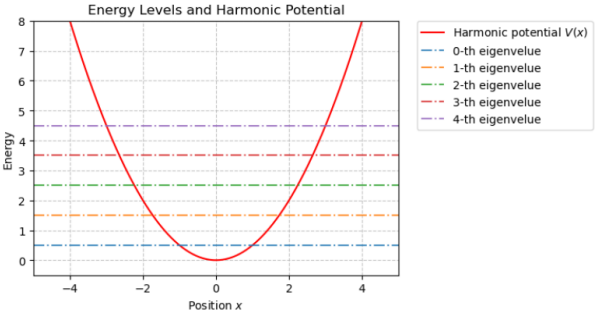

Figure 6: *Values for the first 5 eigenvalues with `order=4`*

From a first qualitative overview, it is possible to see the presence of a great agreement with respect to the analytical results: the numerical eigenvectors seem to resemble the analytical ones, as well as the numerical eigenvalues, which also seem to be equally spaced, as expected.
This gives us the possibility to begin to study in more details the numerical solutions that have been found, focusing our attention in terms of Correctness, Stability, Accuracy, Flexibility and Efficiency.

Let's begin from the Correctness parameter to evaluate the obtained numerical solutions. As explained in the theoretical section, in order to verify the affinity with the numerical solutions we proceed in this way: considering the eigenvalues, we compute the relative errors between the numerical and the analytical eigenvalues; instead, considering the eigenvectors we take the dot product between the numerical and analytical results, and consider $1-|\langle\vec{v}_{num}|\vec{v}_{ana}\rangle|$ as degree of dissimilarity between eigenvectors: the more this number increases, the "more perpendicular" the eigenvectors are. For `order=2` the first $k = 16$ eigenvectors and eigenvalues have been generated and the results, as a function of their index, are displayed in Figure 7:

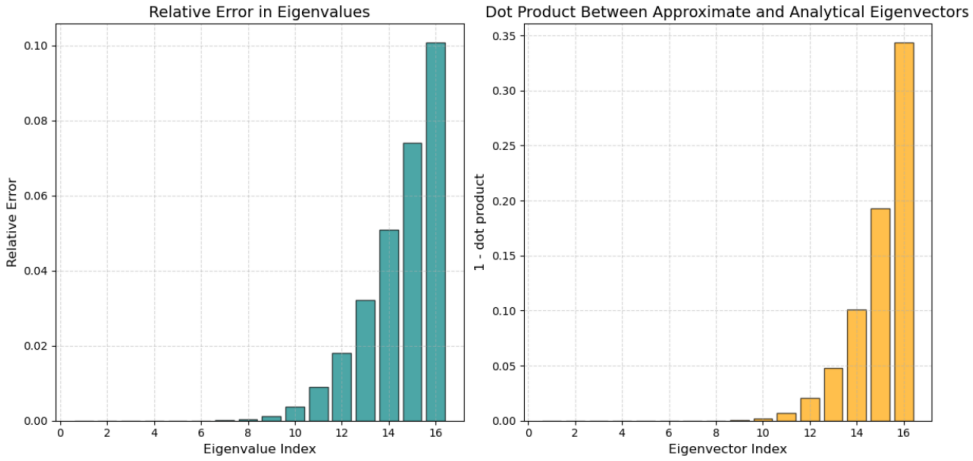

Figure 7: *Correctness of eigenvalues and eigenvectors for* `order=2`

It's possible to notice that as the eigen-index grows up, the errors of both eigenvalues and eigenvectors increases as well, reaching an error around 10% for the $16-$th eigenvalue, while a degree of dissimilarity around 35% for the $16-$th eigenvector. Despite the growth of the error is really fast, one can notice that the errors for the first eigenvalues/eigenvectors are really small, allowing us to conclude that, for `order=2` the numerical solutions can be considered correct within this small error.
A similar analysis has been performed for `order=4` with the goal of analyzing the correctness of higher order corrections, but, even more, with the aim of comparing the results with `order=2` in terms of correctness. For this reason, a similar graph showing the comparison between the two errors have been produced, where the $y-$axis has been put in logarithmic scale in order to better visualize the error difference for the first indices (where it is expected to have discrepancies between the two orders). The results are displayed in Figure 8:

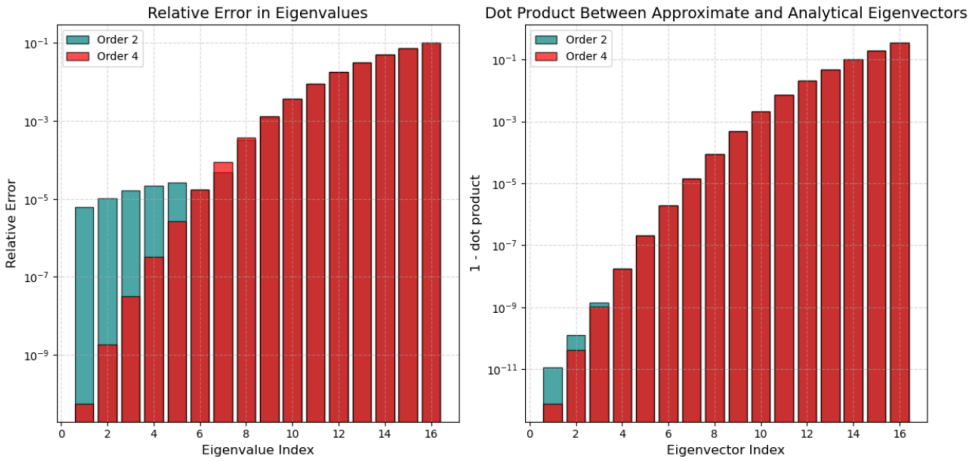

Figure 8: *Correctness of eigenvalues and eigenvectors comparing* `order=2` *and* `order=4`

First of all we still obtained two trends that are compatible and do not vary too much between each other: it means that the choice of the order in this case does not affect too much the simulation in terms of correctness. Secondly, as expected we notice a slight difference for the first indices, having the `order=2` yielding slighly bigger errors than the finer order. However, the errors are still pretty small in both the cases, and below 10^{-3} for the first 10 eigenvalues/eigenvectors.
Now it is possible to start the description of the Stability of our solutions.
Again, we separate the discussion for the 2 different orders, with the additional goal of making a comparison between the two. For this purpose we make use of the function `stability` explained before. In particular, we perform multiple runs in which for each run we compute the eigenvalues/eigenvectors; for the eigenvalues of the same index we compute the standard deviation, while for the eigenvectors of the same index we compute $1 - |\langle \vec{v}_{num} | \vec{v}_{ana} \rangle|$ for all the couples of eigenvalues having the same index belonging to different runs, and we take the mean. Choosing `num_runs=100`, the results for `order=2` are displayed in Figure 9:

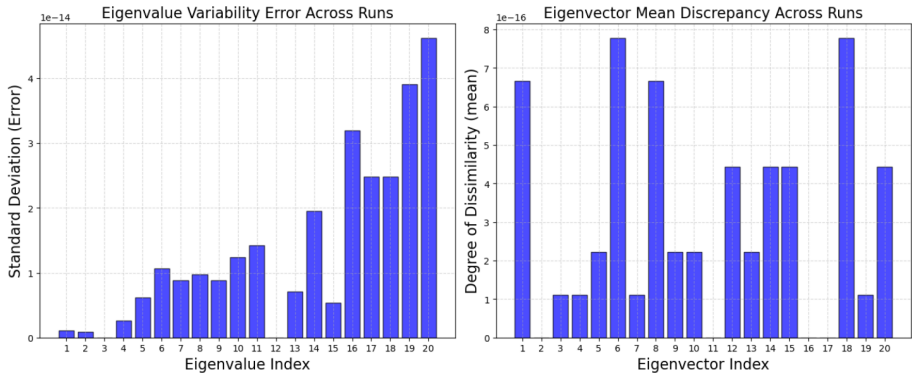

Figure 9: *Stability of eigenvalues and eigenvectors with* `order=2` *and* `num_runs=100`

while the results for `order=4` (still with `num_runs=100`) are displayed in Figure 10:

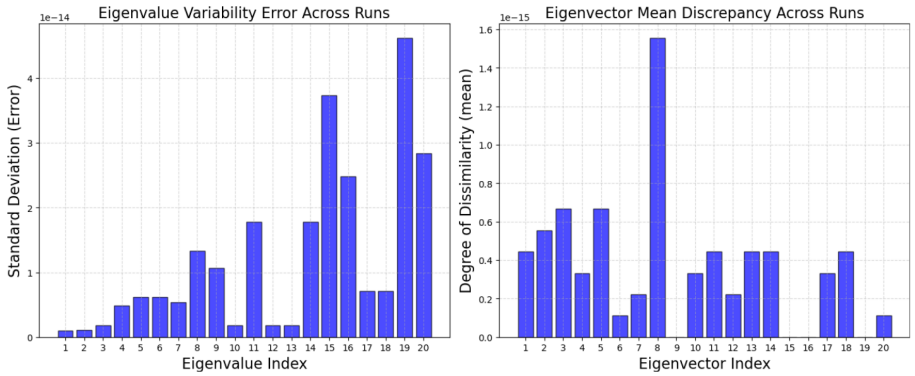

Figure 10: *Stability of eigenvalues and eigenvectors with* `order=4` *and* `num_runs=100`

We can make some considerations. First of all, we can notice that the dissimilarity of eigenvectors/eigenvalues across different runs is in an order of magnitude ranging from 10^{-14} to 10^{-16}: it's a very small value, and probably the observed discrepancy is due to the finite numerical precision used to store the results. In any case, this means that the solutions are very stable for both the orders.
Additionally, taking as hypothesis that the discrepancy is not due to the finite numerical precision, we observe an eigenvalue variability error that increases with the index, even though the trend is not well defined. Instead, considering the eigenvectors, their discrepancy doesn't seem to be correlated with their index, as a much more flat trend emerges.

Finally, comparing the two different orders, one can notice comparable and very similar results in terms of both eigenvalue variability and eigenvector discrepancy, since the order of magnitudes are the same.
Now we can move to a different treatment, describing the problem in terms of Accuracy. Our aim is to analyze the correctness of the solution both scaling the discretization step of the grid and scaling the value of `omega` which determines the curvature of the potential well and influences the spacing of energy levels.
Let's begin by analyzing the correctness of the solution scaling the discretization step of the grid. In particular, we scale from `N_min=100` to `N_max=3000` in steps of `step=100`, and consider `k=16` and `omega=1`.
With `order=2` we results are displayed in Figure 11, where the values are put in logarithmic scale for a better visualization (in this way, the value -2 in the legend, for instance, corresponds to 10^{-2}):

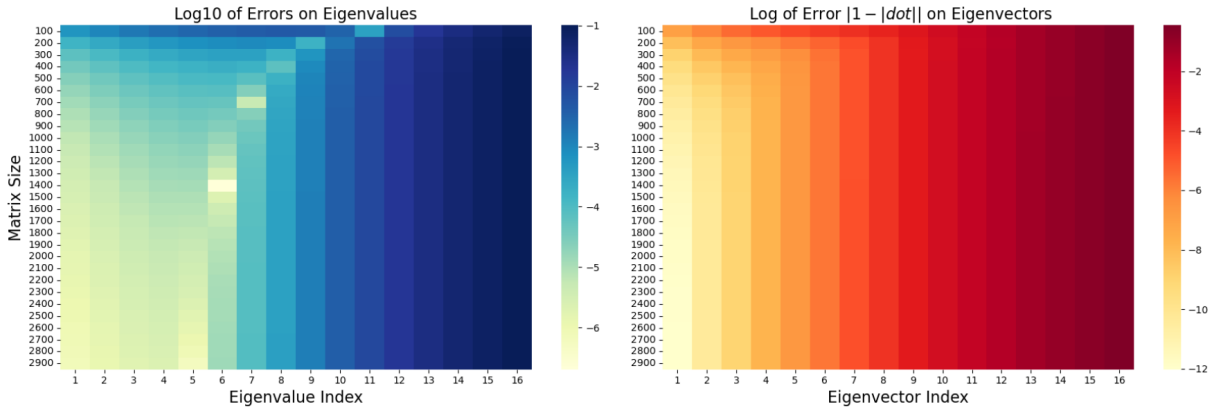

Figure 11: *Correctness of eigenvalues and eigenvectors with `order=2` and varying the matrix size*

The same picture is computed for `order=4` and the results displayed in Figure 12:

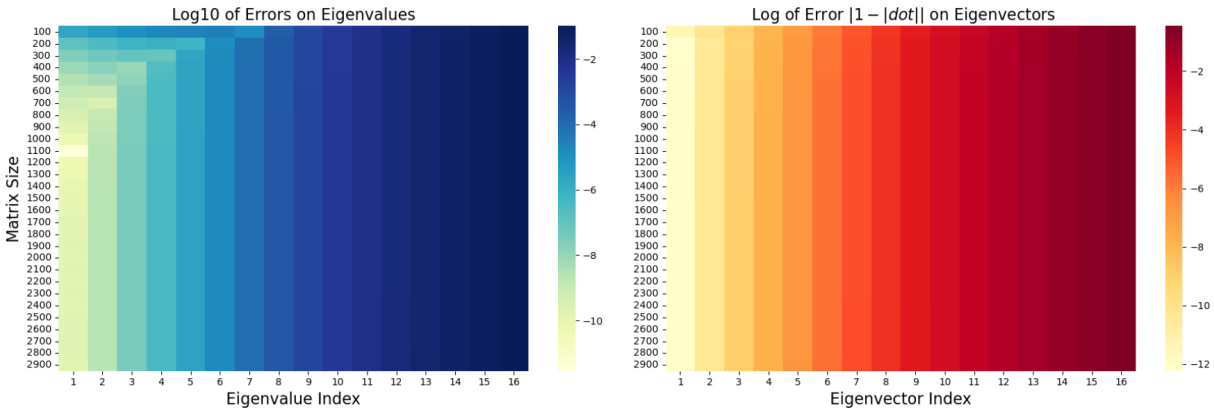

Figure 12: *Correctness of eigenvalues and eigenvectors with `order=4` and varying the matrix size*

From these results similar trends emerge. First of all, the discrepancy with the analytical results increases as the eigenvalue/eigenvector increases, as expected also from the previous results. Additionally, more significant in this context is the fact that the correctness of the solution increases as the matrix size increases (discretization step is shortened). This is a very important result that holds for both the orders, and which allows to determine a region of large matrix size and small eigen-index where the error is very small.
Additionally, one can notice that the heatmaps for the `order=4` display smaller errors, as the order of magnitude increases, meaning that, as expected, the higher order provide more accurate

results. Interestingly, in this case the eigenvector heatmap shows that a discrepancy which seems to be independent from the discretizaion step (with slight differences just for the smallest value). Now we can proceed in a similar way, but instead of varying the discretization step (i.e. the matrix size) we change the value of `omega` from `omega_min=0.1` to `omega_max=5.1` in steps of `omega_step=0.2` and considering `k=16` and `N=1000`.
For sake of simplicity, I report only the results with `order=2` (since analogous results are obtain for the higher order) which are displayed in Figure 13, where the values are again put in logarithmic scale for a better visualization:

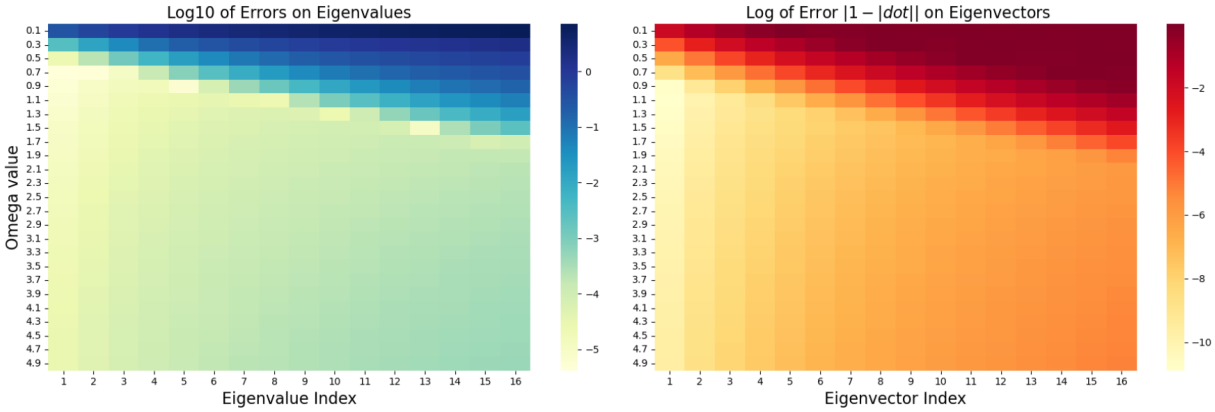

Figure 13: *Correctness of eigenvalues and eigenvectors with `order=2` and varying the `omega` value*

As before, the error grows up as the eigen-index increases; but interestingly, the error goes up as the `omega` value dicreases: having a stronger potential makes the solution more correct, probably due to the fact that the particle is more confined inside the potential, where the discretization works better.
But also, a closer look to the eigenvector heatmap suggests that after we reach a value for `omega` for which the error decreases, if we increase `omega` enough there is a little growth of the error, which highlights a specific region for which the error is minimized.
Just to be clear on this, I fix the eigenvector index to $k = 7$ and display the error as a function of `omega` (basically, I'm taking the column for $k = 7$ in the heatmap); the result can be seen in Figure 14 and confirms the previous analysis.

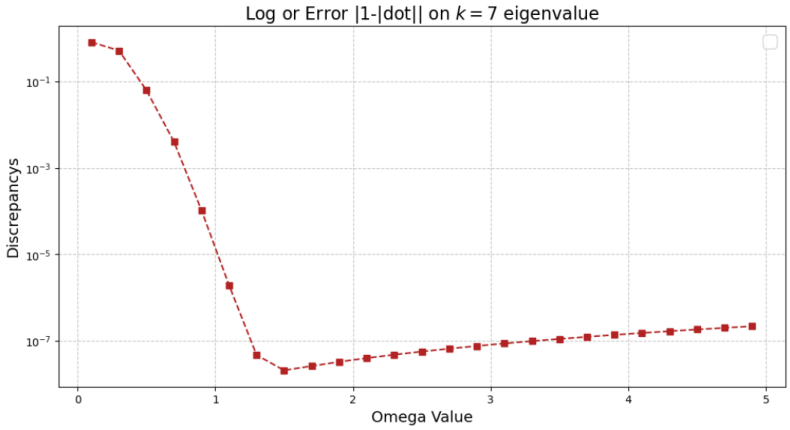

Figure 14: *Correctness of $7-th$ eigenvector with `order=2` and varying the `omega` value*

For the sake of completeness, I evaluated the Accuracy of the discretization, varying the matrix size (hence the discretization step), in terms of stability.

For both orders, I analyzed the stability of the solutions over multiple runs, computing the stability in the exact same way as before. In particular, we scale from `N_min=100` to `N_max=1000` in steps of `step=100`, and consider `k=20`, `omega=5` and `L=5`.
With `order=2` and `num_runs=100` we results are displayed in Figure 15, where the values are put in logarithmic scale for a better visualization:

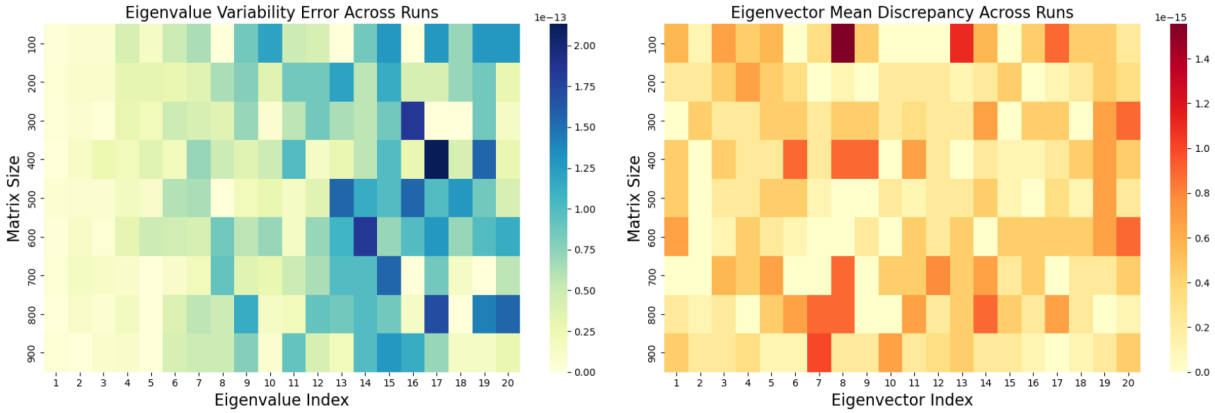

Figure 15: *Stability of eigenvalues and eigenvectors with `order=2` and `num_runs=100` and varying the matrix size*

The same picture is computed for `order=4` and `num_runs=100` and the results displayed in Figure 16:

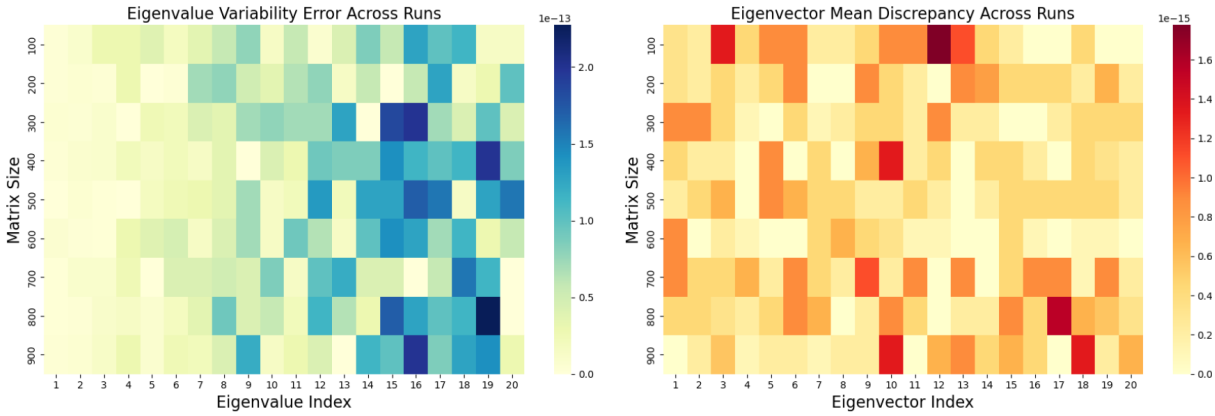

Figure 16: *Stability of eigenvalues and eigenvectors with `order=4` and `num_runs=100` and varying the matrix size*

For both the orders, we can conclude that the instability increases as the eigenvalue index grows up, but seems to be independent from the matrix size. Instead, the stability of eigenvectors does not provide any specific pattern. In any case, the errors due to the instability of the solutions is so small that can be caused by the finite number precision and representation, as said before.
As a final thing we tested the efficiency of the program in computing the whole set of eigenvectors/eigenvalues, in different ways: scaling the discretization step of the grid; scaling `omega`; for different orders.
For this reason I performed a benchmark for the function `hamilton_gen` varying the discretization step and the `omega` value. So, this is basically equivalent of performing a benchmark also for the functions `discretization_size` and `omega_variation` at the same time.
In particular, we scale the discretization step from `N_min=100` to `N_max=3000` in steps of `step=100`, while we scale `omega` from `omega_min=0.1` to `omega_max=5.1`, in steps of

`omega_step=0.2`.
With `order=2` and `L=5` the CPU times for computing the whole set of eigenvalues/eigenvectors are displayed in Figure 17:

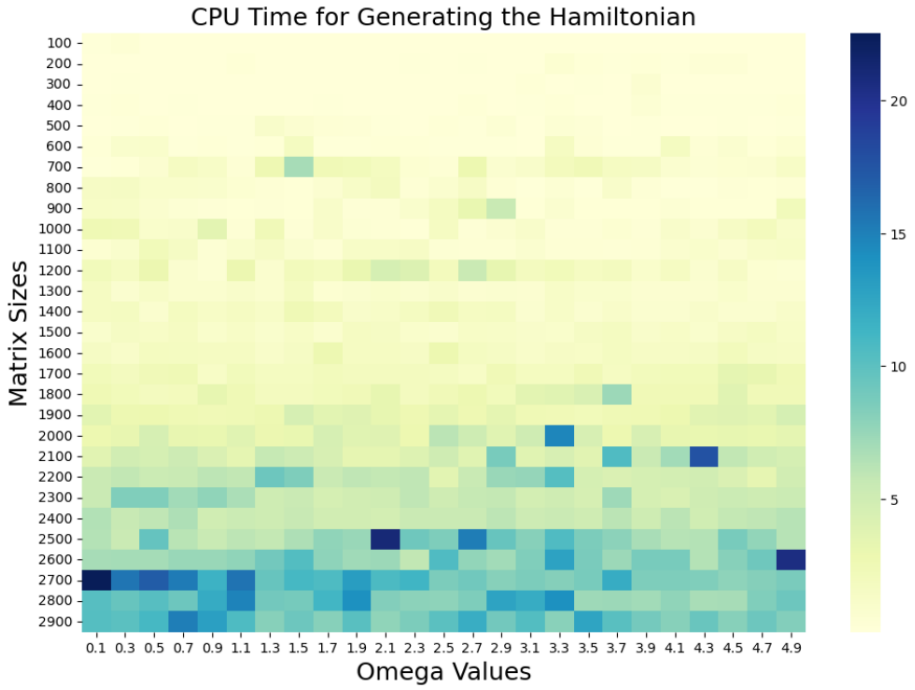

Figure 17: *CPU time for computing the whole set of eigenvalues and eigenvectors, for `order=2` varying the matrix size and `omega`*

The same picture is computed for `order=4` and the results displayed in Figure 18:

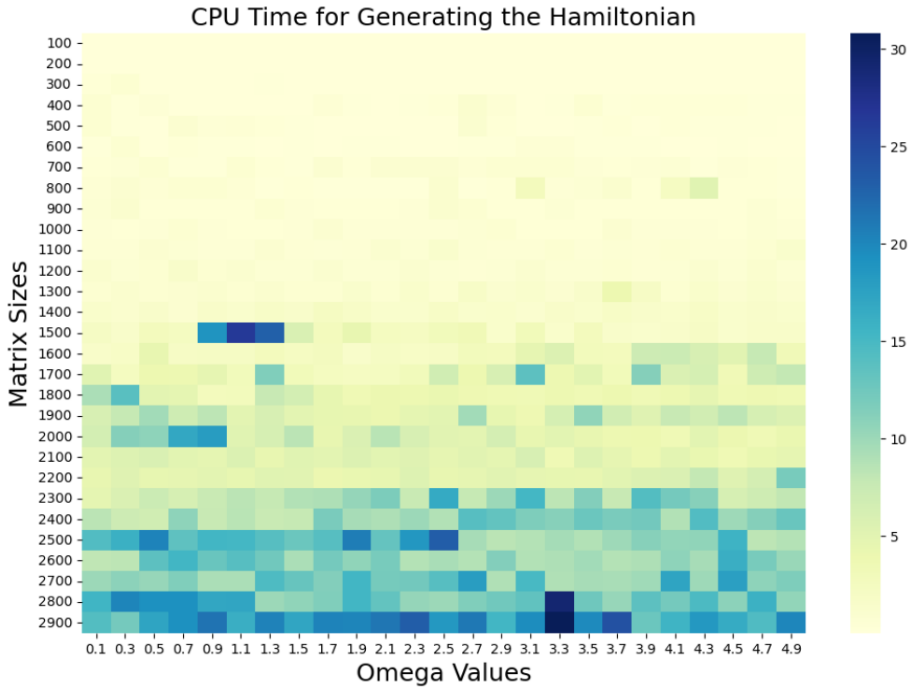

Figure 18: *CPU time for computing the whole set of eigenvalues and eigenvectors, for `order=4` varying the matrix size and `omega`*

For both the orders, the CPU times seem to be independent from the `omega` values, while,

as expected, they increase as the discretization step becomes smaller (i.e. the matrix size increases). Additionally, as can be seen from the legend to the right, the CPU times for the `order=4` seem to be slightly bigger than the lower order ones.

As a final thing, I would like to make some considerations about the employment and application of sparse matrices in this problem, since we are using tridiagonal and pentadiagonal matrices to compute the Hamiltonian.
In particular, for both the orders, we start the analysis by computing the time taken to find the lowest eigenvalue, comparing the performances between the employment of sparse matrices and dense matrices.
For `order=2` the results are shown in Figure 19, while for `order=4` in Figure 20:

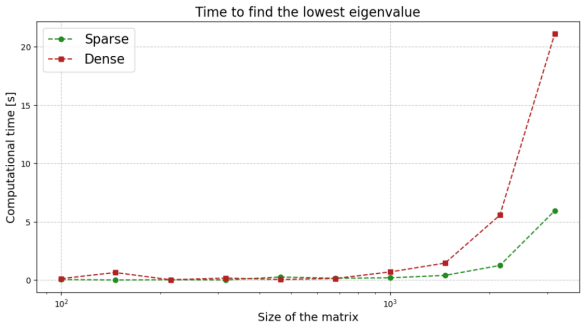

Figure 19: *Time taken to compute the lowest eigenvalue with the Sparse and Dense methods, for* `order=2`

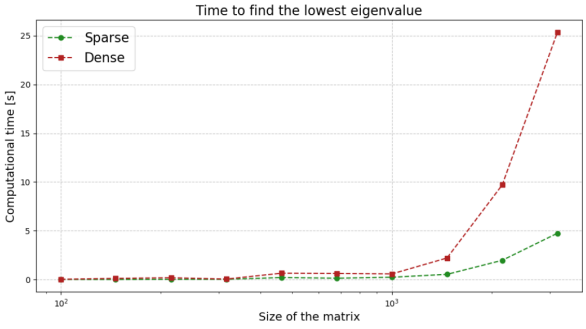

Figure 20: *Time taken to compute the lowest eigenvalue with the Sparse and Dense methods, for* `order=4`

It is possible to see that the CPU time needed to compute the lowest eigenvalue is almost the same until the matrix size starts to approach a value around 10^3: in this moment the two trends start to separate. It means that, for the computation of the lowest eigenvalue, it starts to be convenient to use Sparse matrices when the matrix size starts to be larger than 10^3.
I might be useful to analyze for which number of eigenvalues/eigenvectors k it is convenient to use sparse matrices.
For this reason I created an heatmap varying the number of eigenvalues/eigenvectors k to compute and the matrix size. In each entry of the heatmap I report the difference between the time taken in the Dense case and the time taken in the Sparse case: in this way, positive values indicate the convenience of the Sparse Method, negative numbers the convenience of the Dense Method, and the larger the number in absolute value the stronger the convenience.
For `order=2` and `order=4` the results are displayed in Figure 21:

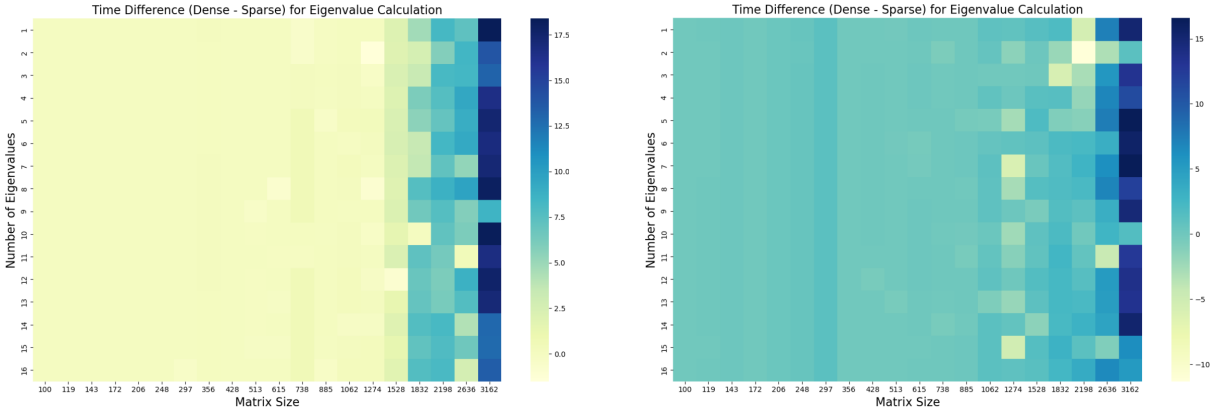

Figure 21: *Time difference between the time in the Dense method and the time in the Sparse method to compute the first k eigenvalues, for* `order=2` *and* `order=4`*, varying the matrix size.*

Note that Figures 19 and 20 correspond to the top horizontal row in the heatmaps.
For the `order=2` the Sparse Method is more convenient starting from a matrix size around 1800, while for `order=4` around 2600. However, the order of magnitude is comparable and further studies need to be focused in this separation region, zooming in and performing a finer analysis.

5 Conclusions

This report has explored the quantum harmonic oscillator by combining analytical solutions with numerical approaches.
A key result of this work is the strong agreement between the numerical and analytical solutions. Using the finite difference method (FDM), we accurately calculated the eigenvalues and eigenvectors for the system. The second-order method produced errors in the eigenvalues that were already small, on the order of 10^{-3}, for the first ten states. The fourth-order method, as expected, delivered even greater accuracy, showing how higher-order schemes can significantly improve precision, at the cost of increased computational effort.
Stability was another important aspect we examined. The numerical solutions were found to be highly stable, with minimal variability between runs. This stability held up across different matrix sizes and eigenvalues, showing that these methods are reliable for a wide range of configurations.
The relationship between accuracy and grid size was also explored in detail. As the number of grid points increased, the numerical solutions became increasingly precise. This result aligns with expectations, as a finer grid better resolves the wavefunctions. However, we also observed that for stronger potentials (higher ω), the particle becomes more localized, making it easier for the numerical grid to capture the system's behavior accurately.
One of the most practical outcomes of this study was the comparison of sparse and dense matrix methods. Sparse methods demonstrated a clear advantage for large matrices, significantly reducing computational time when only a few eigenvalues were needed. For example, sparse solvers were much faster than dense methods for matrix sizes larger than 1000.

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