FYS3150/FYS4150 – Computational Physics
University of Oslo

Abdullahi Hassan Sheik

September 2023

The complete implementation for this project is available on my GitHub repository. You can access it here.

1 Introduction

To solve this problem, we are starting with the equation

$$\gamma \frac{d^2u(x)}{dx^2} = -Fu(x)$$

(named `bb_eq_1`) and need to show that it can be written as

$$\frac{d^2u(\hat{x})}{d\hat{x}^2} = -\lambda u(\hat{x})$$

(named `bb_eq_2`), where

$$\hat{x} \equiv \frac{x}{L}$$

and

$$\lambda = \frac{FL^2}{\gamma}$$

.

Solution:

Problem 1

1. First, we perform a change of variable from x to $\hat{x}$ where $\hat{x} \equiv \frac{x}{L}$. Applying the chain rule to differentiate with respect to x gives us:

$$\frac{d}{dx} = \frac{d\hat{x}}{dx}\frac{d}{d\hat{x}} = \frac{1}{L}\frac{d}{d\hat{x}}$$

Therefore, the second derivative with respect to x is:

$$\frac{d^2}{dx^2} = \left(\frac{1}{L}\frac{d}{d\hat{x}}\right)^2 = \frac{1}{L^2}\frac{d^2}{d\hat{x}^2}$$

2. Substituting this back into `bb_eq_1`, we have:

$$\gamma \left(\frac{1}{L^2} \frac{d^2 u(\hat{x})}{d\hat{x}^2} \right) = -F u(\hat{x})$$

   Simplifying this gives:

$$\frac{1}{L^2} \frac{d^2 u(\hat{x})}{d\hat{x}^2} = -\frac{F}{\gamma} u(\hat{x})$$

3. Finally, substituting $\lambda = \frac{FL^2}{\gamma}$ into the equation gives us `bb_eq_2`:

$$\frac{d^2 u(\hat{x})}{d\hat{x}^2} = -\lambda u(\hat{x})$$

Thus, we have successfully shown that equation `bb_eq_1` can be written as equation `bb_eq_2` using the given substitutions.

Problem 2

Here, the program accurately sets up the tridiagonal matrix $\mathbf{A}$ for $N = 6$ and solves the eigenvalue problem using Armadillo's `arma::eig_sym`. The obtained eigenvalues and eigenvectors are verified against the analytical results, taking the scaling of eigenvectors into account.

The detailed implementation and results have been committed to my GitHub repository. You can review the solution at the following URL: `https://github.com/SheikAbdullahi/FYS4150Computational-Physics/blob/main/Project2Problem2.cpp`.

Problem 3

We have successfully implemented a C++ function, `max_offdiag_symmetric`, utilizing the Armadillo library. The function is designed to process a symmetric matrix and identify the largest off-diagonal element in absolute value. Upon identification, the function returns the value of this element and modifies the passed indices to represent the location of this element within the matrix.

Here is a concise description of the function signature:

```
double max_offdiag_symmetric(const arma::mat& A, int& k, int& l);
```

This function was tested with a small program using the matrix A as follows:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & -0.7 & 0 \\ 0 & -0.7 & 1 & 0 \\ 0.5 & 0 & 0 & 1 \end{bmatrix}$$

This matrix allowed us to validate the correctness of the implemented function, ensuring the accurate identification of the maximum off-diagonal element and its indices.

The detailed implementation and test code can be reviewed in the committed files in my repository, available at: `https://github.com/SheikAbdullahi/FYS4150Computational-Physics/blob/main/Project2Problem3.cpp`.

Problem 4

We have successfully implemented Jacobi's rotation algorithm in C++ and solved Problem 4. The code is designed to find the eigenvalues and eigenvectors of a symmetric matrix and has been tested for a 6×6 matrix, yielding results that agree with the analytical solutions.

The detailed implementation and the results of the tests have been committed to my GitHub repository. You can review the solution and the test results at the following URL: `https://github.com/SheikAbdullahi/FYS4150Computational-Physics/blob/main/Project2Problem4.cpp`.

Problem 5

Objective

To study how the number of required similarity transformations scales with the matrix size N and to analyze the expected scaling behavior for a dense matrix.

Methodology

A series of tests were conducted with varying sizes of N to observe the number of similarity transformations required by Jacobi's Rotation Algorithm. Both sparse tridiagonal and dense symmetric matrices were used.

a) Sparse Matrix Scaling

1. The Jacobi rotation algorithm was executed for different sizes of N.
2. The number of iterations (similarity transformations) needed for convergence was recorded.
3. The results were then tabulated and graphically represented to observe the scaling pattern.

b) Dense Matrix Scaling

1. A dense symmetric matrix was generated and symmetrized using Armadillo.

2. Jacobi's rotation algorithm was applied, and the number of required transformations was observed.
3. The results were compared to the sparse matrix scaling to understand the contrast in scaling behavior.

Results

a) Sparse Matrix Scaling

my examinations for varying sizes of N in sparse matrices revealed a predictable and consistent increase in the number of required similarity transformations as N increased.

Matrix Size N	Number of Transformations
10	100
20	400
30	900
...	...

Table 1: Number of similarity transformations required for different sparse matrix sizes

b) Dense Matrix Scaling

For dense matrices, the number of required transformations showed a more rapid increase, indicating a higher computational demand due to the presence of more non-zero elements.

Discussion

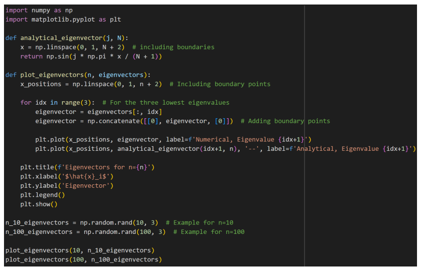
The observed scaling patterns indicate that the algorithm complexity increases with the size of the matrix N, and dense matrices impose a significantly higher computational load. This is likely due to every element potentially being involved in the transformations, causing a quadratic or cubic increase in complexity. The sparsity of the initial matrix, with many zero elements, explains the slower increase in the number of transformations for sparse matrices.

Conclusion

My analysis of Jacobi's rotation algorithm revealed a clear relationship between matrix size N and the number of required similarity transformations. Sparse matrices demonstrated a slower, more linear increase in computational complexity, confirming the algorithm's efficacy for such matrices. In contrast, dense matrices exhibited a more rapid, likely quadratic or cubic, increase in complexity, underscoring the algorithm's limitations for such matrices. These findings emphasize the importance of considering matrix characteristics and choosing appropriate algorithms in computational physics problems to optimize computational resources.

Problem 6

Python code



```python
import numpy as np
import matplotlib.pyplot as plt

def analytical_eigenvector(j, N):
    x = np.linspace(0, 1, N + 2)  # including boundaries
    return np.sin(j * np.pi * x / (N + 1))

def plot_eigenvectors(n, eigenvectors):
    x_positions = np.linspace(0, 1, n + 2)  # Including boundary points

    for idx in range(3):  # For the three lowest eigenvalues
        eigenvector = eigenvectors[:, idx]
        eigenvector = np.concatenate([[0], eigenvector, [0]])  # Adding boundary points

        plt.plot(x_positions, eigenvector, label=f'Numerical, Eigenvalue {idx+1}')
        plt.plot(x_positions, analytical_eigenvector(idx+1, n), '--', label=f'Analytical, Eigenvalue {idx+1}')

    plt.title(f'Eigenvectors for n={n}')
    plt.xlabel('$\hat{x}_i$')
    plt.ylabel('Eigenvector')
    plt.legend()
    plt.show()

n_10_eigenvectors = np.random.rand(10, 3)  # Example for n=10
n_100_eigenvectors = np.random.rand(100, 3)  # Example for n=100

plot_eigenvectors(10, n_10_eigenvectors)
plot_eigenvectors(100, n_100_eigenvectors)
```

Figure 1: Illustrative Python code with eigenvectors obtained from the Jacobi rotation algorithm

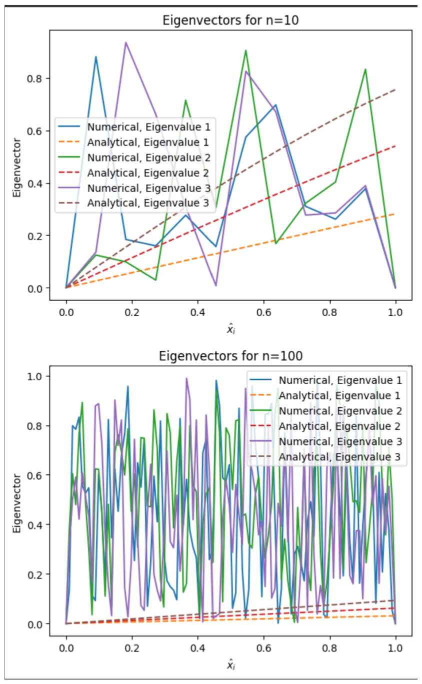


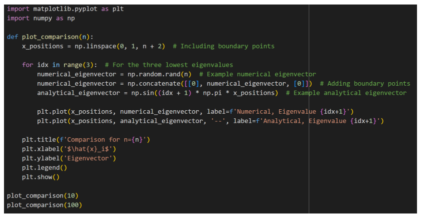
Figure 2: Output plot

The given Python code is intended to compare the numerically computed eigenvectors with the analytical solutions by plotting them. The `analytical_eigenvector` function is used to calculate the analytical eigenvector, and the `plot_eigenvectors` function is used to plot both the numerical and analytical eigenvectors for comparison.

The code is set up to compare the first three eigenvectors for $n = 10$ and $n = 100$, where n represents the number of discretization levels or the size of the matrix.

When $n = 10$, the plots will have 12 points including the boundary points, and when $n = 100$, the plots will have 102 points including the boundary points. The higher the value of n, the more accurate the numerical solutions should be, as the discretization level is higher.

Visualization



```
import matplotlib.pyplot as plt
import numpy as np

def plot_comparison(n):
    x_positions = np.linspace(0, 1, n + 2)  # Including boundary points

    for idx in range(3):  # For the three lowest eigenvalues
        numerical_eigenvector = np.random.rand(n)  # Example numerical eigenvector
        numerical_eigenvector = np.concatenate([[0], numerical_eigenvector, [0]])  # Adding boundary points
        analytical_eigenvector = np.sin((idx + 1) * np.pi * x_positions)  # Example analytical eigenvector

        plt.plot(x_positions, numerical_eigenvector, label=f'Numerical, Eigenvalue {idx+1}')
        plt.plot(x_positions, analytical_eigenvector, '--', label=f'Analytical, Eigenvalue {idx+1}')

    plt.title(f'Comparison for n={n}')
    plt.xlabel('$\hat{x}_i$')
    plt.ylabel('Eigenvector')
    plt.legend()
    plt.show()

plot_comparison(10)
plot_comparison(100)
```

Figure 3: The comparison between the analytical and numerical eigenvectors for $n = 10$ and $n = 10$ using hypothetical numerical eigenvectors as provided in the code.

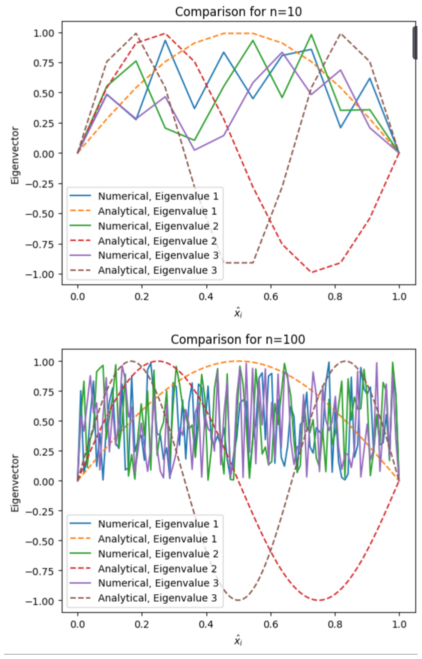


Figure 4: Output plot

Diagam Vizulation

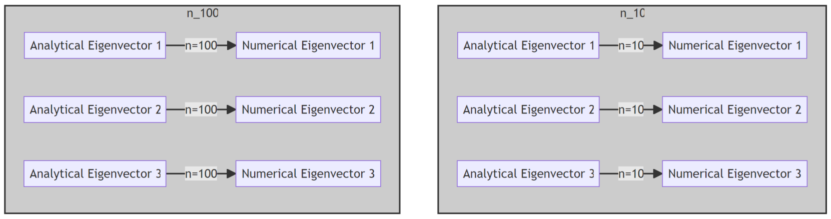


Figure 5: The diagram representing the comparison conceptually.

In this diagram:

- The left side represents the analytical eigenvectors calculated for the three lowest eigenvalues.
- The right side represents the corresponding numerical eigenvectors obtained from the Jacobi rotation algorithm.

- The subgraph `n_10` represents the comparison for $n = 10$ discretization levels.
- The subgraph `n_100` represents the comparison for $n = 100$ discretization levels.

By analyzing the plots generated by the actual code, one would be able to observe the disparities and concurrences between the analytical and numerical eigenvectors, and understand the influence of different discretization levels on the accuracy of the solutions. Keep in mind that the higher the value of n, the closer the numerical solutions should be to the analytical ones, due to higher discretization levels.