Project 2

For the course FYS3150

Erik Grammeltvedt, Erlend Tiberg North and Alexandra Jahr Kolstad

September 30, 2019
Week 37 - 40

Contents

Abstract: short motivation and presentation of the results and the findings

Introduction: you want to motive the reader about the problem and why you want solve it

Theory: explaining the theory behind the solution method and the problem

method/implementation: how you implement the solution in order to fix/solve the problem

results/graphs/tables: presenting the results

discussion: Discussing the result from previous section

conclusion: concluding the findings, your neutral opinion, etc... and future work

appendix:How you derived your method, theory, etc...

- How many similarity transformations are needed before you reach a result where all non-diagonal matrix elements are essentially zero? *Besvart i discussion*
- Try to estimate the number of transformations and extract a behavior as function of the dimensionality of the matrix. *Besvart i discussion*
- Compare your results with the analytical ones. *Besvart i discussion og noe i results*
- Comment your results (here you could for example compute the time needed for both algorithms for a given dimensionality of the matrix). *Besvart i discussion og noe i results*
- Try to figure out other tests your code should pass, based either on the mathematical properties of the algorithms or more program specific tests. Implement at least two unit tests in this project. *Besvart i koden, men tenker å kommentere hvorfor valgte disse i method*
- Study the results as functions of the number of integration points N and your approximation to ρ max. The analytical results with our scaling for the one-electron energies are $\lambda = 3, 7, 11, 15, \ldots$.
- How many integration points do you need in order to reproduce the analytical results with say four leading digits after the decimal point?
- Comment the results for the lowest state (ground state) as function of varying strengths of ω r.

1 Abstract

bla bla bla bla bla bla bla

2 Introduction

All programs are found at our GitHub-repository.

Hvilke filer som er til hvilke oppgaver

Our project consists of the files `jacobimethod.cpp` and `plot_data.py`. For exercises **b)** through **e)** we use the file `jacobimethod.cpp`. Also for exercise **b)** we have the file `plot_data.py`.
This article is seperated into sections with some subsections.

3 Theory

3.1 Orthogonality of a unitary transformation

Firstly (?) we are going to prove that $\vec{w}_i = U\vec{v}_i$ is an orthogonal or unitary transformation that preserves the dot product and orthogonality. We start by multiplying $\vec{w}_j^T$ with $\vec{w}_i$ to take the vector product, also called the dot product. If the vector product of these vectors is equal to δ_{ij}, given by $\vec{v}_j^T \vec{v}_i = \delta_{ij}$ in the exercise, then the dot product and orthogonality is preserved. In this exercise we assume that $U^T U = I$, where I is the identity matrix, because this defines a unitary matrix U which we compute with in this exercise.

The vector product is calculated as followed:

$$\begin{aligned}
\vec{w}_j^T \vec{w}_i &= (U\vec{v})^T U\vec{v}_i \\
&= \vec{v}_j^T U^T U \vec{v}_i \\
&= \vec{v}_j^T \vec{v}_i \\
&= \delta_{ij}
\end{aligned}$$

The vector product of $\vec{w}_j^T$ and $\vec{w}_i$ is δ_{ij}, which proves that the dot product and orthogonality is preserved for the transformation.
In this prject we compute with a symmetric matrix, similar to the matrix **A** in project 1. This matrix is given by the matrix equation

$$\begin{bmatrix}
d & a & 0 & \dots & 0 & 0 \\
a & d & a & \dots & 0 & 0 \\
0 & a & d & \dots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & a & d & a \\
0 & 0 & 0 & 0 & a & d
\end{bmatrix}
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix}
= \lambda
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix}$$

where $d = \frac{2}{h^2}$ and $a = -\frac{1}{h^2}$. We implement these values later in exercise **d)**. For now in this exercise, we have set $d = 2$ and $a = -1$. **still correct??**

λ are eigenvalues given by the equation

$$\lambda_j = d + 2a\cos\left(\frac{j\pi}{N+1}\right) \tag{1}$$

given for $j = 1, 2, ..., N$.

4 Method

Kommentere hvorfor vi bruker Jacobi method til å løse egenpar-problemene.

In the Appendix we have proven that $\vec{w}_i = U\vec{v}_i$ preserves the orthogonality for unitary transformations. This is an important result because it shows that by using the Jacobi method the eigenvalues and eigenvectors is preserved, regardless of how many transformations, or iterations, is performed. Therefore we use the Jacobi method to find the eigenpairs, because in quantum mechanics the eigenpairs are an excellent way of illustrating for instance a harmonic oscillator.

Kommentere hvorfor valgte de testene som ble valgt.

The first unit test in our program is a test to confirm that the largest non-diagonal element indeed is the largest non-diagonal element. This test is important for the Jacobi method to function properly, as the algorithm is based on firstly finding the biggest non-diagonal element and rotating the matrix to decrease the value of this element. When the value is smaller than a set tolerance, the algorithm finds the new largest non-diagonal element in this new matrix and rotates. Eventually all non-diagonal elements will be smaller than the tolerance and we have the desired matrix. If the algorithm is incapable of finding the largest non-diagonal element, then the rotation will result in a wrong matrix and wrong eigenpairs.

The second unit test in our program is a test to confirm that a known simple matrix returns the same and correct eigenvalues. By using the equation 1 we can easily compute the eigenvalues of our matrix, and then compare this with the eigenvalues found in our algorithm. This test is important for confirming that our eigenpair solver works correctly and that we get the correct results.

5 Results

Our results are as shown in the Appendix. We also have `.txt`-files for all the raw data generated by the projects up on GitHub.

Under follows the data in `stats.txt`.

```
n, iterations, timespan eig_sym, timespan ours
3, 10, 1.137520e-04, 1.666000e-05
5, 32, 8.632200e-05, 6.015200e-05
10, 154, 8.277800e-05, 3.797710e-04
15, 363, 1.395090e-04, 1.351192e-03
20, 644, 2.044150e-04, 3.183269e-03
25, 1025, 2.189260e-04, 7.074493e-03
30, 1463, 1.089399e-02, 1.494651e-02
40, 2685, 1.947745e-02, 4.040845e-02
50, 4115, 8.989085e-03, 8.897613e-02
60, 6007, 7.917790e-04, 1.650397e-01
70, 8081, 1.022618e-02, 2.948578e-01
80, 10487, 1.577296e-03, 4.732270e-01
90, 13338, 2.079035e-03, 7.397605e-01
100, 16438, 1.184104e-02, 1.103670e+00
110, 19905, 2.845618e-03, 1.580353e+00
120, 23547, 3.005831e-03, 2.222943e+00
130, 27615, 3.534705e-03, 2.997598e+00
```

```
140, 31981, 4.432247e-03, 4.003528e+00
150, 36537, 4.577840e-03, 5.204313e+00
160, 41531, 5.158932e-03, 6.712134e+00
170, 47005, 6.375916e-03, 8.584403e+00
180, 52424, 7.180886e-03, 1.065379e+01
190, 58289, 7.780051e-03, 1.319042e+01
200, 64379, 8.258574e-03, 1.607514e+01
```

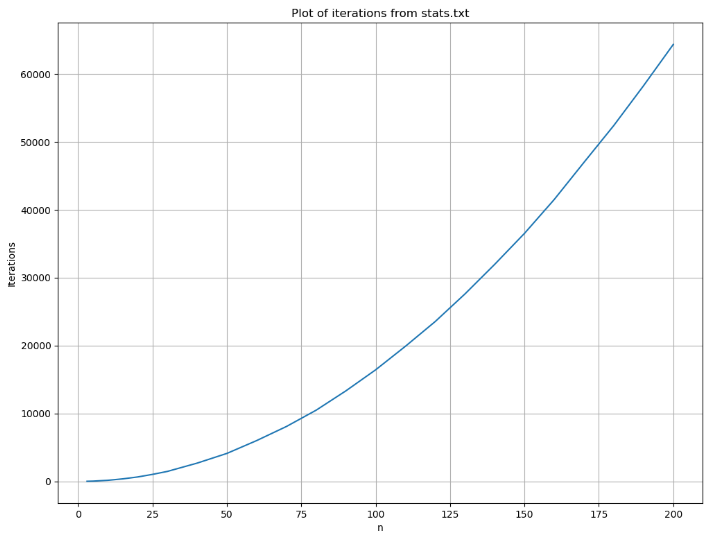

Figure 1: The plot of iterations for the Jacobi method as function of the dimension n of the matrix $\mathbf{A}$.

Figure (1) shows that the number of iterations as a function of the dimension n of the matrix has an exponential increase. This means that for larger values of n, we need many similarity transformations for our matrix to have all non-diagonal elements become zero. This result coincides with the time difference in the algorithms. We observe that for small matrix-dimensions n our algorithm is sligthly faster than the Armadillo function `eig_sym`. When n increases in value, the time used in our algorithm increases exponentially when looking at figure (2). For the biggest given dimension, $n = 200$, our algorithm uses 16s while `eig_sym` uses only 0.008s. Here we can observe how slow our algorithm is compared to `eig_sym`.

ferdig med å kommentere dette?

6 Discussion

The number of similarity transformations, also called iterations, needed to reach the desired matrix depends on the dimension n. For instance a run of our matrix $\mathbf{A}$ given as a (10×10) matrix, there are 154 transformations needed. This number is only exact for this specific run, as it will change for any differences to the matrix, both size and elements.

In the lecture notes it states that for the Jacobi method there is no way to predict the number of transformations needed. See this file under *Discussion for Householder's method for eigenvalues.* **Riktig??**

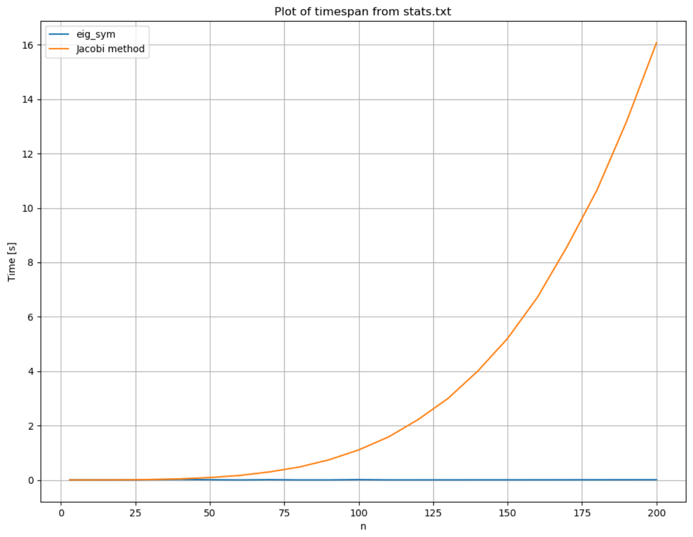

Figure 2: The plot of the time the function `eig_sym` from Armadillo uses and the time Jacobi method uses as functions of the dimension n of the matrix $\mathbf{A}$.

The convergence rate of the Jacobi method is however poor, one needs typically 3n2 - 5n2 rotations and each rotation requires 4n operations, resulting in a total of 12n3 - 20n3 operations in order to zero out non-diagonal matrix elements. Fra lecture notes.

The Jacobi method is considerably slower for large values of n mainly because the matrix is larger, which means that the algorithm has more elements to rotate and because it has to execute more iterations. The slowness for the Jacobi method is based on when n increases, the algorithm increases the value of some elements, while it decreases the value of others. Consequences of this is that the algorithm has to compute multiple iterations compared to `eig_sym` for the same values of n. Because there is a time difference in the algorithms, we know that `eig_sym` does not use the Jacobi method, and is therefore conseiderably faster. The function most likely observes that the matrix is tridiagonal and finds the easiest solution, based on if-else-statements and different types of eigenpair solvers. When looking at the definition of `eig_sym` given on this website, the function has an argument `method` for describing two different methods for using the function. The default option is `dc`, which stands for *divide-and-conquer*, while the other option is `std`, which stands for *standard*. The `dc` method is sligthly faster for smaller matrices, dimension $\tilde{3}00$ and smaller, but for larger matrices it is considerably faster. When testing `eig_sym` with both methods for matrices of sizes $n = \{1000, 5000, 10000\}$ there is already a notable diffrence for $n = 1000$, where `dc` is ten times faster than `std`. Because we compute with matrices of dimension $n = 200$ and smaller, the difference in usage of `eig_sym` can be ignored as it gives very similar results.

Koden bruker lengre tid både fordi matrisen er større, slik at det er mer å rotere og fordi den må utføre flere iterasjoner.

Hvorfor er jacobimethod tregere enn eigsym? økende n, jacobi øker verdien til noen elementer, mens den minker noen andre verdier. altså den fucker noen verdier, mens den fikser noen andre.

eigsym bruker ikke jacobi metoden, og kan dermed være raskere. den kan f.eks. bruke thomas algorithm, som vi programerte i project 1, som er raskere for større matriser. tror den bruker en if-else-statement med ulike algoritmer til å løse de forskjellige matrisene - har prøvd å finne kildefilen gitt ved armadillo, men den finner jeg ikke.

når man skriver inn std istedenfor dc for eigsym er dc litt raskere for store matriser, med n rundt 200, men siden vi driver med relativt små matriser vil man ikke merke stor forskjell. for større matriser, altså 10 i 3 og litt høyere merker man stor forskjell
eigsym ser at den er tridiagonal og finner letteste løsningsmetode

7 Conclusion and perspective

8 Appendix

8.1 `jacobimethod.cpp`

Skal kommentere noe for egenverdi og egenvektor?
Commenting the code `jacobimethod.cpp`. OBS.

The program starts with defining a function for finding the max values of the off-diagonal elements. This is the function `offdiag`, which is taken from the lecture notes. The same applies for the function `Jacobi_rotate`. `Jacobi_rotate` is the function for rotating and computing the matrix. It calculates the equation

$$\mathbf{B} = \mathbf{S}^T A \mathbf{S}$$

for finding the diagonal matrix with the eigenvalues of the matrix **A**. It also computes the eigenvectors, and stores them in a matrix **R**.

In the main function we define the matrix **A** given by the constants d and a. See ? for the definitions of the matrix. Then the clock is set to start, and it times how long the armadillo

8.2 `plot_data.py`

The program `plot_data.py` reads the `.txt`-file made in `jacobimethod.cpp` and plots the data. `jacobimethod.cpp` generates the file `stats.txt`, which contains the dimension of the matrices, `n`, the number of iterations, `i`, the time used for the Armadillo-funtion `eig_syl`, `timespan eig_sym` and the time used in our algorithm, `timespan ours`. `plot_data.py` plots the number of iterations needed given by different values of the matrix dimension and it plots the time needed as a function of the matrix dimensions. The values in `stats.txt` is taken from multiple runs of `jacobimethod.cpp` for different dimensions of the matrices. The figures (1) and (2) are the plots from `plot_data.py`. **HAR FLERE PLOT Å PLOTTE**

Skal egentlig kommentere dette?????

9 References

Link to the PDF for Project 2.

Our GitHub-repository.

Link to lecture slides in FYS3150 - Computational Physics.

Offical Armadillo website for documentation of all contents in the library.

Analytical results for specific oscillator frequencies.