Project 2

For the course FYS3150

Erik Grammeltvedt, Erlend Tiberg North and Alexandra Jahr Kolstad

October 4, 2019
Week 37 - 40

Contents

1 Abstract **2**

2 Introduction **2**

3 Theory **3**

3.1 Orthogonality of a unitary transformation . . . 3

3.2 The Jacobi method . . . 3

3.3 The problem in this project . . . 5

3.3.1 Classical problem: buckling beam . . . 5

3.3.2 Quantum mechanical problem: harmonic oscillator . . . 6

4 Method **7**

5 Results **7**

6 Discussion **9**

7 Conclusion and perspective **10**

8 Appendix **10**

8.1 Assorted data . . . 10

8.2 `plot_data.py` . . . 10

9 References **11**

1 Abstract

The overall goal of this project is to make an eigenvalue solver and to show its relevance by solving some quantum mechanical problems, but firstly by testing it for a simpler classical problem. In this project we are focusing on using the Jacobi method in order to find the eigenvalues and eigenvectors of a matrix. As well as comparing it to imported eigenvalue and eigenvector solvers from armadillo to see witch method is most potent.

2 Introduction

All programs are found at our GitHub-repository.

The versatility of many numerical methods are astounding and fundamentally important for humanities understanding of the world. For instance the method used in this problem starts out very basic but goes on to solve problems in the quantum mechanic realm with small variations to the code. This project focuses on numerical eigenvalue solvers or more specifically the Jacobi method. The article starts by describing how the one dimension beam problem can be simplified and discretized into a numerically solvable problem.

$$-\lambda u_i = \frac{-(-u_{i-1} + 2u_i - u_{i-1})}{h^2}$$

The problem in this project is divided into two parts: first we are going to look at a beam in the y-direction and secondly electrons in a three-dimensional harmonic potential. This is respectively a classical problem and a quantum mechanical problem. Both of these problems are eigenvalue problems, and will be solved with the Jacobi method implemented in our algorithm. We will compare the results from our algorithm with results of the Armadillo function `eig_sym`.

Followed by the proof that the Jacobi method is useable because it does not change the eigenvectors, the project then explains basic quantum mechanics needed in order to see how the Jacobi method is applied in this project. Then an explanation to how we have implemented the Jacobi method is given in the program `jacobomethod.cpp`. Followed by the results showing that the Jacobi method is very slow compared to other eigenvalue and eigenvector solving methods that can be used, from for instance Armadillo. The conclusion to this project is that the Jacobi method is an inefficient method to solve eigenvalues, however the project demonstrates the power of such a solver and its many different applications.

This project is seperated into sections with some subsections. See table of contents for page number to the sections. In Theory we will present essential theory to understand our problem, why we solve it and how we solve it. In Method we will explain why we use the Jacobi algorithm to solve our problem. Furthermore it includes explanations to the unit tests we chose in our algorithm. In Results we will present the data and the plots generated in our programs, with a short explanation of what the results present. The section Discussion is where we will consider our problem, and discuss why the two methods Jacobi method and the Armadillo function `eig_sym` gives different computation times. After this we will conclude our findings in the section Conclusion and perspective. Finally we have the section Appendix for additional information, such as derivations, comments and figures, tables or text, and the section References for citing every reference in one place.

Our project consists of the files `jacobimethod.cpp` and `plot_data.py`. `jacobimethod.cpp` is the main program for computing the Jacobi method, while `plot_data.py` plots the data generated in `jacobimethod.cpp`, which is appended into `.txt`-files. The `.txt`-files in this project are `stats.txt`, `harmonic.txt` and `qdot.txt`.

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 $U^TU = I$, 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 $a = -\frac{1}{h^2}$ or $d = 2$ and $a = -1$ depending on if the problem is a quantum mechanical problem or a classical problem.

λ are eigenvalues given by the equation

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

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

3.2 The Jacobi method

The Jacobi method uses the concept described in Orthogonality of a unitary transformation in order to generate the eigenvalues and eigenvetors. This is done by multiplying by the Jacobi matrix and the transpose of the Jacobi matrix. The Jacobi matrix is given as:

$$\begin{bmatrix} 1 & a & 0 & \dots & 0 & 0 \\ a & 1 & a & \dots & 0 & 0 \\ 0 & a & cos(\theta) & \dots & 0 & sin(\theta) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & a & 1 & a \\ 0 & 0 & -sin(\theta) & 0 & a & cos(\theta) \end{bmatrix}$$

It performs an orthogonal transformation of the matrix **A** based on the following equation

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

As shown above this will not change the orthogonality of the eigenvectors. This means that we can perform this mathematical operation as many times as we want without changing the equation.

Then the metod is reapeated until the matrix is diagonalized. In order to make the Jacobi method most effective the Jacobi matrix needs to be made in such a way that it zeroes out the biggest number. However, it will also change values other than the diagonalelements. This leads to an uncertainty whether or not the values on the diagonal becomes smaller or bigger as we run through the method many times. However by taking the sum of all the non-diagonal elements in **A** and comparing it to the sum of all the non-diagonal elements in **B** we see that $\sum \mathbf{A} > \sum \mathbf{B}$. So if we repeat this sequence many times one should get that the non-diagonal elements in the matrix approaches zero. The sum of a matrix is given as

$$\sum \mathbf{A} = \sqrt{a_{ij}}^2 \tag{2}$$

Through algebra we can show that:

$$\sum \mathbf{B} = \sum \mathbf{A} - K \tag{3}$$

K contains values of different elements in **A**, leaving us with a total value on the non-diagonal elements i B.
In order to calculate the angle we must ensure that the sin non-diagonal elements become zero. Therefore we calculate θ accordingly

$$\cot(2\theta) = \tau = \frac{a_{ii} - a_{kk}}{a_{kl}} \tag{4}$$

We now use s for sin, t for tan and c for cos.

This leaves us with the quadratic equation:

$$t^2 + 2\tau t - 1 = 0 \tag{5}$$

giving

$$t = -\tau \pm \sqrt{1+\tau^2} \tag{6}$$

$$c = \frac{1}{\sqrt{1t^2}} \tag{7}$$

$$s = tc \tag{8}$$

This is the theory behined the Jacobi method, which is the numerical eigenvalue solver that will be used in this project. This method can be used to solve a big variety of problems, such as harmonic osillation in quantum mechanics.

3.3 The problem in this project

3.3.1 Classical problem: buckling beam

In this project we start with the a beam which is moving up and down in the y-direction. This problem can be broken down into this equation

$$\gamma \frac{d^2u(x)}{dx^2} = -Fu(x) \tag{9}$$

The length of the beam x is $[0, L]$ and the force acting on $(L, 0)$ to bring it back to its original position is F. The displacement for the beam is given by $u(x)$. γ is a constant that takes into account different properties of the beam, like the rigidity of the beam.

The equation (9) is now found and the process of making a numerical problem starts. In order to get a start and an end point with the same value, we say that $u(0) = u(L) = 0$. We now introduce the length defining value ρ:

$$\rho = \frac{x}{L} \tag{10}$$

Given our definition of the lengths we now have that $\rho = [0, 1]$. Meaning that we can substitute ρ for x in equation (9). By rearranging the matrix, we get the following equation:

$$\frac{d^2u(\rho)}{dx^2} = \frac{FL^2}{R} = -\lambda u(\rho) \tag{11}$$

This problem can be discretized by saying that the double derivative of the function is equal to drawing a line between two points. When the distance between the two points becomes infinetly small we get the derivative:

$$u'' = \frac{u(\rho_i + h) - 2u(\rho_i) + u(\rho_i - h)}{h^2} + O(h^2) \tag{12}$$

From earlier the min and max values of ρ are given as $\rho_{min} = 0$ and $\rho_{max} = 1$. For the discretization this means that h becomes the step length going from ρ_{min} to ρ_{max} where $\rho_{min} = \rho_0$ and ρ_{max} is equal to the total number of steps used, N. This leaves us with:

$$h = \frac{\rho_{min} - \rho_{max}}{N} \tag{13}$$

This allows ρ to be defined as:

$$\rho_i = \rho_0 + ih \tag{14}$$

Now all the tools needed to solve the eigenvalue problem has been acquired and the eigenvalues can be solved through this equation:

$$\lambda u_i = \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} \tag{15}$$

3.3.2 Quantum mechanical problem: harmonic oscillator

As the theory behind the quantum mechanics is not the main part of this paper, but rather to show the relevance of numeric eigenvalue solvers, we will therefore only provide the relevant parts of the theory here.

The quantum mechanical problem is descirbed as electrons moving harmoniously in three dimensions confined by a potential barrier. We assume repulsion between the electrons as a results of Coulombs law. Also we think of the electrons as spheres.
The radial part of the Schrödinger equation for one electron is:

$$\frac{\hbar}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr} - \frac{l(l+1)}{r^2}\right)R(r) + V(r)R(r) = ER(r) \tag{16}$$

Through algebraic manipulation we get the differential equation

$$\frac{d^2}{d\rho^2}u(\rho) + \rho^2 u(\rho) = \lambda u(\rho) \tag{17}$$

With the addition of the potential $V_i = \rho_i^2$ we get

$$\lambda u_i = \frac{-(-u_i - 1 + 2u_i - u_i - 1)}{h^2} + \rho_i^2 u(\rho_i) \tag{18}$$

This is the equation we solve with the Jacobi method for one electron in a harmonic oscillator potential.

Now looking at the combined force between two harmonic oscillating electrons we get the Schrödinger equation:

$$\frac{\hbar}{2m}\frac{d^2}{dr^2}u(r) + \frac{1}{2}kr^2 u(r) = Eu(r) \tag{19}$$

By using algebraic manipulation the equation can be rewritten as

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \omega p^2\psi(p) + \frac{1}{\rho} = \lambda u(p) \tag{20}$$

The same numeric eigenvalue solver can be used here as for the beam problem with some minor adjustments.

4 Method

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.

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

We have `.txt`-files for all the raw data generated by the projects up on GitHub.

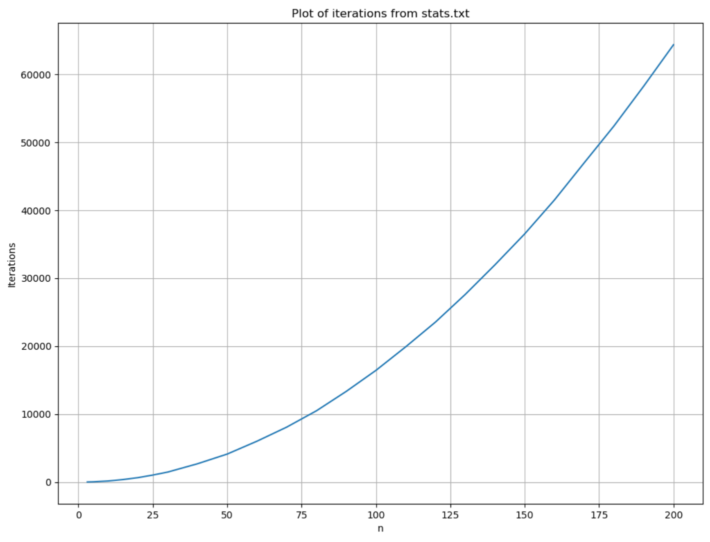

Figure 1: The plot of iterations for the Jacobi method as function of the dimension n of the matrix $\mathbf{A}$.

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

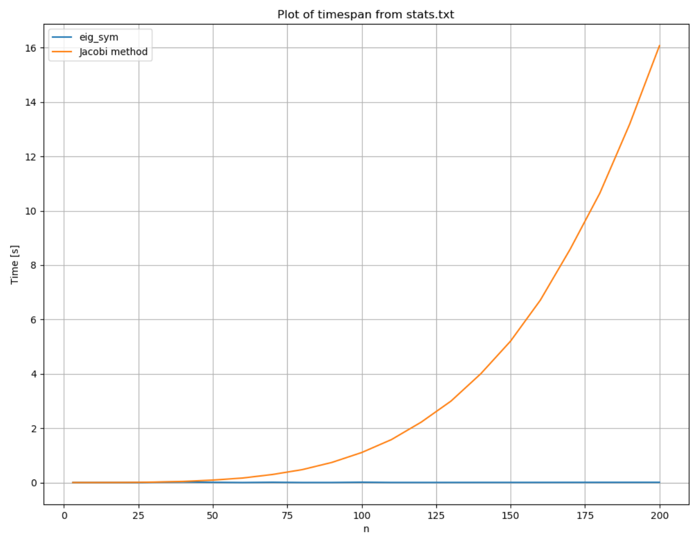

Figure 2: The plot of the time the function `eig_sym` from Armadillo uses and the time Jacobi method uses as functions of the dimension n of the matrix $\mathbf{A}$.

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`.

Table 1: The eigenvalues of the harmonic oscillator for one electron for different integration points N and different dimensionless variables ρ_{max}.

ρ_{max}		1	5	10
	5	(10.11, 36.49, 72.49, 108.49)	(3.21, 9.07, 14.08, 20.31)	(3.49, 25.72, 45.17, 70.17)
	50	(10.17, 39.77, 88.92, 157.47)	(3.23, 7.32, 11.38, 15.43)	(3.45, 7.62, 11.70, 15.70)
	100	(10.16, 39.80, 89.10, 158.05)	(3.11, 7.17, 11.20, 15.23)	(3.23, 7.33, 11.39, 15.42)
N	150	(10.16, 39.80, 89.13, 158.16)	(3.08, 7.11, 11.14, 15.16)	(3.15, 7.22, 11.27, 15.30)
	200	(10.16, 39.80, 89.14, 158.20)	(3.06, 7.08, 11.10, 15.12)	(3.11, 7.17, 11.20, 15.23)
	300	(10.15, 39.80, 89.15, 158.22)	(3.04, 7.06, 11.07, 15.09)	(3.08, 7.11, 11.14, 15.16)
	400	(10.15, 39.80, 89.15, 158.23)	(3.03, 7.04, 11.05, 15.07)	(3.06, 7.08, 11.10, 15.12)

Table 2: The eigenvalues of the harmonic oscillator for two electrons where $N = 400$ and $\rho_{max} = 400$ for varying ω_r.

ω_r	Eigenvalues
0.01	(0.310621, 0.679419, 1.2189, 1.94162)
0.5	(12.3107, 42.6021, 92.3392, 161.689)
1	(4.08621, 7.96089, 11.8888, 15.8406)
5	(7.9323, 37.8442, 57.8296, 77.8307)

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.*
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 considerably 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.

We can see that changing the frenquency has an impact on the energy of the eigenstates. We can first confirm that our algorithm works by running for $\omega_r = 0.25$, which gives us 1.25. This is twice that of the tabulated value in the article by M. Taut. They have another scaling which gives them half of ours.

By running for $\omega_r = \{0.01, 0.5, 1, 5\}$ we can see the difference in values (see 5). We generally have an increasing energy for the ground state (0.3, 12.3, 4.1, 7.9). This makes sense as a higher frequency generally leads to increased energy (think of electromagnetic waves for example). The 12.3 is from $\omega_r = 0.5$ and kind-of sticks out. We can imagine that this frecuency resonates weirdly with the system, perhaps in a poor way, giving us the increased energy.

All results in table (5) have a number of integration points that the analytical results do not posess. In addition our algorithm is incapable of giving the analytic results with four leading digits. This could be from an errror in the indexing, or perhaps a weird compiler in Windows QtCreator. Nevertheless the algorithm gives correct results to an acceptable accuracy.

7 Conclusion and perspective

Through testing different size matrices the data shows that the Jacobi method is inferior to the methods used by the Armadillo function. What this difference comes from is hard to say for certainty because it was close to impossible to find the methods the Armadillo function used. What can be said for certain is that Jacobi method changes in this case two elements to zero, but also has an effect on other elements in the matrix. A zero element can be changed to a non-zero element. Therefore the method needs to be repeated many times in order to make each element within the epsilon-range of 0. Despite the methods ineffectiveness it did enough to solve all the quantum mechanics problems and one of the advantages of the Jacobi method is that it can solve the eigenvectors of any given matrix. All in all it is a consistent method for solving eigenvalue problems, but the lacking efficiency makes it less viable for big projects where other methods have clearly shown higher potency.

8 Appendix

8.1 Assorted data

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
```

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_sym`, `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 used 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`.

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.