FYS4150, Computational Physics
Project 2

Andreas Bjurstedt

September 30, 2019

Abstract

Jacobi's method for finding the eigenvalues and their corresponding eigenvectors of a symmetric matrix **A** is presented and programmed in Python. The programmed algorithm is tested on three physical problems where their equations are scaled: A beam exposed to a compressive force which causes buckling, an electron moving in a three dimentional potential well, and two electrons moving in the well resulting in an interacting Coulomb potential between them. We assume spherical symmetry in the two quantum mechanical problems. Tested against the exact analytical solutions, the Jacobi algorithm finds the eigenvalues of the buckling beam problem with a high degree of accuracy. Due to the varying values on the matrix diagonal in the two electron-well problems, a higher number of grid points are required for the numerical eigenvalues to close in towards the exact analytical values. Finding the eigenvectors is beyond the scope of this project

When we compare the Jacobi algorithm with the eigvals(**A**) function from the SciPy library, we see that the Jacobi algorithm doesn't find the eigenvalues of the matrix **A** in a time efficient way.

Introduction

In this project we present Jacobi's method for finding the eigenvalues and their corresponding eigenvectors of a symmetric matrix **A**. The method for finding the eigenvalues is programmed as a function in Python. Finding the corresponding eigenvectors is beyond the scope of this project. We test the algorithm on three physical eigenvalue problems.

Physical problem 1 is a horizontal beam prevented from moving in vertical direction in its end points. A compressive force acting on one of its end points causes buckling of the beam. In physical problem 2, one electron moves in a three dimensional potential well, and in problem 3 two interacting electrons are moving in the well. The interaction between the two electrons is due to the Coulomb potential. We assume spherical symmetry in these two quantum mechanical problems. Each of the three problem's equations are scaled in order to be without physical dimensions. Then we are able to see beyond their different physical parameters and see the similarities in their mathematical expressions. Approximating the second derivative in these equations as we did in project 1, see [2], using a finite number of grid points, leads to a matrix eigenvalue system $\mathbf{Av} = \lambda\mathbf{v}$. Jacobi's method finds the eigenvalues λ of the problem by diagonalizing the matrix $\mathbf{A}$. The diagonal in the diagonalized matrix then consists of the problem's eigenvalues.

When the scaled buckling beam problem is tested against exact analytical eigenvalue, Jacobi's method showed great accuracy in calculating the eigenvalues in this problem for as few as 6 grid points (endpoints not included). For the two scaled electron well problems, a higher number of grid points are required for the numerical eigenvalues to close in towards the exact analytical values. The reason for this is that the matrix elements on the diagonal of $\mathbf{A}$ doesn't have the same value. They vary due to the potential term in these two problems, which separate their scaled equations from each other and from the scaled equation of the buckling beam.

The Jacobi algorithm doesn't find the eigenvalues in a time efficient way compared to the SciPy library's eigvals($\mathbf{A}$) function, when tested against eachother for an increasing number of grid points in the buckling beam problem. When n in the matrix size $n \times n$ of matrix $\mathbf{A}$ is doubled, the number of similar transformations required in order to diagonalize the matrix is multiplied by approximately four. When the matrix size reach $n \times n = 160 \times 160$ the Jacobi method already starts to spend noticeable time in finding the eigenvalues.

Methods

Methods, part 1: Jacobi's method for finding the eigenvalues and their corresponding eigenvectors of a symmetric matrix.

The eigenvalues λ and the corresponding eigenvectors $\mathbf{x}$ of an $n \times n$ matrix $\mathbf{A}$ is defined by

$$\mathbf{Ax} = \lambda\mathbf{x}$$

Here we present the Jacobi's metod to find the eigenvalues λ of a symmetric matrix $\mathbf{A}$ numerically. The method is presented in the lecture notes found in [1]. Before we start, we need some results from linear algebra.

First a definition:
When $\mathbf{S^{-1}} = \mathbf{S^T}$ so that $\mathbf{S^TS} = \mathbf{SS^T} = I$, we call $\mathbf{S}$ an orthogonal matrix.

We state result 1 without proof:
Let $\mathbf{A}$ be a symmetric $n \times n$ matrix, were all the matrix elements a_{ij}, $i, j = 1, 2, \cdots, n$ are real numbers. Then there exist a real orthogonal $n \times n$ matrix $\mathbf{S}$ so that

$$\mathbf{S^TAS} = \mathbf{D}$$

where

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & 0 & & \cdots & 0 \\ 0 & \lambda_2 & 0 & & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & & 0 & \lambda_{n-1} & 0 \\ 0 & \cdots & & 0 & 0 & \lambda_n \end{bmatrix}$$

Each eigenvalue λ_i isn't necessary unique. Two or more eigenvalues can have the same value.

The proof of result 2 is given in the lecture notes in [1]:
$\mathbf{A}$ is a symmetric $n \times n$ matrix, were all the matrix elements
$a_{ij}, \quad i, j = 1, 2, \cdots, n$ are real numbers. When $\mathbf{S}$ is an orthogonal matrix, we call $\mathbf{B} = \mathbf{S^TAS}$ a similarity transformation of $\mathbf{A}$. The eigenvalues λ_i doesn't change during the transformation, so the eigenvalues of $\mathbf{B}$ is the same as the eigenvalues of $\mathbf{A}$. The corresponding eigenvectors $\mathbf{x_i}$ do change however.

Even though the main focus in this project is about finding the eigenvalues of a symmetric matrix, we also include a result regarding the corresponding eigenvectors. We will prove this one.

Result 3:
An orthogonal transformation preserves the orthogonality of the eigenvectors.

Proof:
Lets consider a basis of vectors $\mathbf{v_i}, \quad i = 1, 2, \cdots, n$, where

$$\mathbf{v_i} = \begin{bmatrix} v_{i1} \\ \cdots \\ \cdots \\ v_{in} \end{bmatrix}$$

Assume that the basis is orthogonal, that is $\mathbf{v_j^T v_i} = \delta_{ij}$. An orthogonal transformation $\mathbf{w_i} = \mathbf{S v_i}$ preserves the dot product, because

$$\mathbf{w_i} \cdot \mathbf{w_j} = (\mathbf{S v_i}) \cdot (\mathbf{S v_j}) = (\mathbf{S v_i})^{\mathbf{T}} (\mathbf{S v_j}) = \mathbf{v_i^T S^T S v_j} = \mathbf{v_i^T v_j} = \mathbf{v_i} \cdot \mathbf{v_j}$$

When the transformation preserves the dot product, it also preserves the orthogonality.

We are now ready to present the Jacobi's method in order to find the eigenvalues of the symmetric $n \times n$ matrix $\mathbf{A}$. The idea is to apply subsequent similarity transformations so that

$$\mathbf{S_N^T} \cdots \mathbf{S_2^T S_1^T A S_1 S_2} \cdots \mathbf{S_N} = \mathbf{D}$$

In our algorithm we don't pursue the perfect diagonal matrix $\mathbf{D}$. Instead we require that the norm of the off diagonal matrix elements in the symmetric matrix $\mathbf{D}$ is

$$\text{off}(\mathbf{D}) = \sqrt{\sum_{i=1}^{n} \sum_{i=1, j \neq i}^{n} a_{ij}^2} > 10^{-8}$$

in order for it to continue further with subsequent similarity transformations. We base $\mathbf{S^T}$ on the orthogonal matrix

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$$

which rotates 2×1 vectors clockwise with an angle θ in their plane. For simplicity, we illustrate a similar transformation step on a symmetric 3×3 matrix $\mathbf{A}$. We assume that the largest non diagonal matrix element in $\mathbf{A}$ is $|a_{23}|$. In the symmetric matrix $\mathbf{B} = \mathbf{S^T A S}$ we want $b_{23} = 0$.

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

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & b_{23} \\ b_{13} & b_{23} & b_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{bmatrix}$$

Element by element we then get

$$\begin{aligned} b_{11} &= a_{11} \\ b_{12} &= a_{12}c - a_{13}s \\ b_{13} &= a_{13}c + a_{12}s \\ b_{22} &= a_{22}c^2 - 2a_{23}cs + a_{33}s^2 \\ b_{33} &= a_{33}c^2 + 2a_{23}cs + a_{22}s^2 \\ b_{23} &= (a_{22} - a_{33})cs + a_{23}(c^2 - s^2) \end{aligned}$$

We can now find the rotation angle θ which gives $b_{23} = 0$. With θ we also have the expressions for b_{12}, b_{13}, b_{22} and b_{33}, and are ready for a new similarity transformation finding the rotation angle which set the largest of the remaining non zero, non diagonal matrix elements to zero.

In general: We assume that the largest non diagonal matrix element in $\mathbf{A}$ is $|a_{kl}|, \quad k < l$. In the symmetric matrix $\mathbf{B} = \mathbf{S^T A S}$ we want $b_{kl} = 0$. Element by element we get

$$\begin{aligned} b_{ii} &= a_{ii} \\ b_{ik} &= a_{ik}c - a_{il}s \\ b_{il} &= a_{il}c + a_{ik}s \\ b_{kk} &= a_{kk}c^2 - 2a_{kl}cs + a_{ll}s^2 \\ b_{ll} &= a_{ll}c^2 + 2a_{lk}cs + a_{kk}s^2 \\ b_{kl} &= (a_{kk} - a_{ll})cs + a_{kl}(c^2 - s^2) \\ i &= 1, \cdots, n, \quad \text{where } i \neq k, l \end{aligned}$$

In order for $b_{kl} = 0$,

$$(a_{kk} - a_{ll})cs + a_{kl}(c^2 - s^2) = 0$$

That means

$$\begin{aligned}(a_{ll}-a_{kk})cs &= a_{kl}(c^2-s^2)\\ \frac{a_{ll}-a_{kk}}{a_{kl}} &= \frac{c^2-s^2}{cs}\end{aligned}$$

Multiplying the numerator and denominator on the right hand side with $1/cos^2(\theta) = 1/c^2$, we get

$$\frac{a_{ll}-a_{kk}}{a_{kl}} = \frac{1-t^2}{t}$$

where $t = tan(\theta)$. We also have the relation

$$2\tau = 2cot(2\theta) = \frac{1-t^2}{t}$$

That means

$$2\tau = \frac{a_{ll}-a_{kk}}{a_{kl}} = \frac{1-t^2}{t}$$

So θ has to fulfill the second order equation

$$tan^2(\theta) + 2cot(2\theta)tan(\theta) - 1 = 0$$

where the value of $2\tau = cot(2\theta)$ is given above. We choose $tan(\theta)$ with the lowest absolute value $|tan(\theta)|$ from

$$tan(\theta) = -cot(2\theta) \pm \sqrt{cot^2(2\theta)+1}$$

to find

$$\begin{aligned}c &= cos(\theta) = \frac{1}{\sqrt{1+tan^2(\theta)}}\\ s &= sin(\theta) = tan(\theta)cos(\theta)\end{aligned}$$

Knowing the rotation angle θ which gives $b_{kl} = 0$, we can also find the other matrix elements in **B** which the similar transformation changes.

A short recap applying the Jacobi's method in order to diagonalize and then finding the eigenvalues of an $n \times n$ symmetric matrix **A**:

- Find the largest non diagonal element $|a_{kl}|, \quad k < l$ in the symmetric matrix A.

- Find $t = tan(\theta)$ and then calculate $c = cos(\theta)$ and $s = sin(\theta)$ of the rotation angle θ which give $b_{kl} = 0$ in the similar transformation $\mathbf{B} = \mathbf{S^T A S}$.
- Update the other matrix elements which changes due to the similar transformation: $b_{kk}, b_{ll}, b_{kl} = b_{lk}, b_{ik} = b_{ki}$ and $b_{il} = b_{li}, \quad i = 1, \cdots, n, \quad \text{where } i \neq k, l$.
- Continue with subsequent similar transformations as long as norm of the off diagonal elements in the symmetric matrix $\mathbf{B}$ is

$$\text{off}(\mathbf{B}) = \sqrt{\sum_{i=1}^{n} \sum_{i=1, j\neq i}^{n} a_{ij}^2} > 10^{-8}$$

Although finding the corresponding eigenvectors $\mathbf{x}$ to the eigenvalues λ of the symmetric matrix $\mathbf{A}$ is outside the scope of this project, the method to find them is presented here. In order to find diagonal matrix $\mathbf{D}$ containing the eigenvalues λ on the diagonal, the matrix system $\mathbf{Ax} = \lambda\mathbf{x}$ undergoes N similarity transformations:

$$\begin{aligned} \mathbf{D} &= \mathbf{S_N^T} \cdots \mathbf{S_2^T S_1^T A S_1 S_2} \cdots \mathbf{S_N} \\ \mathbf{I} &= \mathbf{S_N^T} \cdots \mathbf{S_2^T S_1^T} [\mathbf{x_1}, \mathbf{x_2} \cdots, \mathbf{x_n}] \end{aligned}$$

where the coloums in the $n \times n$ identity matrix $\mathbf{I}$ contains the transformed eigenvectors corresponding to the eigenvalues on the diagonal in $\mathbf{D}$. As we have seen, the similar transformations don't change the eigenvalues, but the eigenvectors are changed. The orthogonality between the eigenvectors are preserved though. In order to find the eigenvectors of the original matrix system, we just reverse the transformations

$$[\mathbf{x_1}, \mathbf{x_2} \cdots, \mathbf{x_n}] = \mathbf{S_1 S_2} \cdots \mathbf{S_N I} = \mathbf{I S_1 S_2} \cdots \mathbf{S_N}$$

As a simple illustration, lets go back to our system with a symmetric 3×3 matrix $\mathbf{A}$ where we wanted $b_{23} = 0$ after the similar transformation $\mathbf{B} = \mathbf{S^T A S}$. On three real column vectors this particular similar transformation will give

$$\begin{aligned}
[\hat{\mathbf{x}}_\mathbf{1}, \hat{\mathbf{x}}_\mathbf{2}, \hat{\mathbf{x}}_\mathbf{3}] &= [\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}]\mathbf{S} \\
\begin{bmatrix} x_{11} & x_{12}c - x_{13}s & x_{12}s + x_{13}c \\ x_{21} & x_{22}c - x_{23}s & x_{22}s + x_{23}c \\ x_{31} & x_{32}c - x_{33}s & x_{32}s + x_{33}c \end{bmatrix} &= \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{bmatrix}
\end{aligned}$$

For the two column vectors the similar transformation changes, we get

$$\begin{aligned}
\hat{x}_{i2} &= x_{i2}c - x_{i3}s \\
\hat{x}_{i3} &= x_{i2}s - x_{i3}c, \quad i = 1, 2, 3
\end{aligned}$$

In general where we want $b_{kl} = 0$ in the symmetric $n \times n$ matrix $\mathbf{A}$, we get

$$\begin{aligned}
\hat{x}_{ik} &= x_{ik}c - x_{il}s \\
\hat{x}_{il} &= x_{ik}s - x_{il}c, \quad i = 1, 2, \cdots, n
\end{aligned}$$

This step is performed for each of the similar transformations $\mathbf{S_1 S_2 \cdots S_N}$

Methods, part 2: Three eigenvalue problems

The Jacobi's method for finding eigenvalues are tested on three physical problems:

Problem 1:
The problem and the numerical discretization is described in the project 2 description, see [1]. A summary follows here. A horizontal beam in 2D has length L and is prevented from moving in vertical direction in its end points. It is exposed to a horizontal compressive force F in its right end, which will cause the beam to buckle. The equation is scaled and the problem without physical dimensions is

$$\frac{d^2u(\rho)}{d\rho^2} = -\frac{FL^2}{R}u(\rho) = -\lambda u(\rho), \quad u(0) = u(1) = 0$$

R is a constant parameter including material properties and cross section geometric properties of the beam. In order to find a numerical solution we use the same discrete aproximation to the second derivative as we did in project 1 (see the project 1 description in [1] and the project 1 report in [2]). The approximate discretization of the problem with $n+2$ discrete points and the index $i = 1, 2 \cdots n$ then is

$$-\frac{u(\rho_i + h) - 2u(\rho_i) + u(\rho_i - h)}{h^2} = \lambda u(\rho_i), \quad u(\rho_0 = 0) = u(\rho_{n+1} = 1) = 0$$

where the distance between the discrete points is

$$h = \frac{\rho_{n+1} - \rho_0}{n+1} = \frac{1}{n+1}$$

and $\rho_i = \rho_0 + ih = ih$. In a more compact form we get

$$-\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = \lambda v_i, \quad v_0 = v_{n+1} = 0$$

which when written out leads to the matrix system

$$\begin{bmatrix} d & a & 0 & 0 & \cdots & 0 & 0 \\ a & d & a & 0 & \cdots & 0 & 0 \\ 0 & a & d & a & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & \cdots & a & d & a \\ 0 & \cdots & \cdots & \cdots & 0 & a & d \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \cdots \\ v_{n-1} \\ v_n \end{bmatrix} = \lambda \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \cdots \\ v_{n-1} \\ v_n \end{bmatrix}$$

where $d = 2/h^2$ and $a = -1/h^2$. λ is the eigenvalue and $\mathbf{v}$ is the corresponding eigenvector.

Problem 2:
The problem and the numerical discretization is described in the project 2 description, see [1]. A summary follows here. An electron moves in a three dimensional oscillator potential well. We assume spherical symmetry and only looks at the radial part of Schroedinger's equation. For the discrete electron energy states E_{nl} we set the quantum number $l = 0$ The equation is scaled and the problem without physical dimensions is

$$-\frac{d^2u(\rho)}{d\rho^2} + \rho^2 u(\rho) = \lambda u(\rho), \quad u(0) = u(\infty) = 0$$

The scaled equation resembles the scaled equation for the beam in problem 1. In addition we have an extra term involving the harmonic oscillator potential $V = \rho^2$. An approximate discretization of the problem with $n+2$ discrete points and index $i = 1, 2 \cdots n$ is

$$-\frac{u(\rho_i+h)-2u(\rho_i)+u(\rho_i-h)}{h^2}+\rho_i^2u(\rho_i)=\lambda u(\rho_i),$$
$$u(\rho_0=0)=0 \quad \text{and} \quad u(\rho_{n+1}=\rho_{max})=0$$

Since we can't set $\rho_{n+1}=\infty$, we have used $\rho_{n+1}=\rho_{max}$ instead, where we can adjust the value of ρ_{max}. Then

$$h=\frac{\rho_{n+1}-\rho_0}{n+1}=\frac{\rho_{max}}{n+1}$$

and $\rho_i=\rho_0+ih=ih$. In a more compact form we get

$$-\frac{v_{i+1}-2v_i+u_{i-1}}{h^2}+\rho_i^2v_i=\lambda v_i, \quad v_0=v_{n+1}=0$$

which when written out leads to the matrix system

$$\begin{bmatrix} d_1 & a & 0 & 0 & \cdots & 0 & 0 \\ a & d_2 & a & 0 & \cdots & 0 & 0 \\ 0 & a & d_3 & a & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & \cdots & a & d_{n-1} & a \\ 0 & \cdots & \cdots & \cdots & 0 & a & d_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \cdots \\ v_{n-1} \\ v_n \end{bmatrix} = \lambda \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \cdots \\ v_{n-1} \\ v_n \end{bmatrix}$$

where $a=-1/h^2$ and $d_i=2/h^2+\rho_i^2$.

Problem 3:
The problem and the numerical discretization is described in the project 2 description, see [1]. A summary follows here. Two electrons now moves in a three dimensional oscillator potential well, and they interact due to the repulsive Coulomb interaction between them. Again we set $l=0$. We assume spherical symmetry and only looks at the radial part of Schroedinger's equation, which now is dependent on the relative coordinate $r=r_1-r_2$ between the two electrons and the center of mass coordinate $R=1/2(r_1+r_2)$. By separation of variables, we can look at the r dependent part of the Schroedinger equation alone. The equation is scaled and the problem without physical dimensions is

$$-\frac{d^2\psi(\rho)}{d\rho^2}+\left(\omega_r^2\rho^2+\frac{1}{\rho}\right)\psi(\rho)=\lambda\psi(\rho), \quad \psi(0)=\psi(\infty)=0$$

This scaled equation is quite similar to the scaled equation for one electron. The difference is the potential $\omega_r^2\rho^2+1/\rho$, which has replaced ρ^2. The frequency ω_r is explained in the project 2 description. We can then use the same matrix equation as for the one-electron problem by only replacing the diagonal elements $d_i = 2/h^2 + \rho_i^2$ with $d_i = 2/h^2 + \omega_r^2\rho_i^2 + 1/\rho_i$. We use Jacobi's method with $\rho_{max} = 10$, instead of $\rho_{max} = 5$ which we used in the one electron problem, to find the eigenvalues of the system.

Results

Jacobi's method is programmed in Python in order to find the eigenvalues of the discrete representations of the three physical problems presented.

Problem 1:
For the beam exposed to a compressive force which causes the beam to buckle, we compare our results with the analytical solution for the eigenvalues given i the description of project 2

$$\lambda_i = d + 2acos\left(\frac{i\pi}{n+1}\right), \quad i = 1, 2, \cdots, n$$

and with a function, eigvals(A), from the SciPy library, see [3] used for finding eigenvalues of a matrix A. The 6 eigenvalues for the 6×6 matrix system for the buckling beam problem are shown in table (1). We see from the table that in this case the Jacobi's method is accurate for at least 6 leading digits after the decimal point. For each eigenvalue we could calculate the corresponding compressive force

$$F = \frac{R\lambda}{L^2}$$

and find the corresponding eigenvector which gives the deformation pattern of the scaled beam problem for that particular force. Notice that since the buckling beam problem is a continous problem by its nature, a higher number of discrete points n would give a finer discrete division of the eigenvalues: The eigenvalues would be closer to each other.

λ_i	Analytical	Python	Jacobi
i = 1	9.70505095E+00	9.70505095E+00	9.70505095E+00
i = 2	3.68979994E+01	3.68979994E+01	3.68979994E+01
i = 3	7.61929485E+01	7.61929485E+01	7.61929485E+01
i = 4	1.19807052E+02	1.19807052E+02	1.19807052E+02
i = 5	1.59102001E+02	1.59102001E+02	1.59102001E+02
i = 6	1.86294949E+02	1.86294949E+02	1.86294949E+02

Table 1: A 6×6 matrix system for the buckling beam problem : The analytical eigenvalues, the eigenvalues calculated with the SciPy function eigvals(A) and the eigenvalues calculated with Jacobi's method.

We also measure the elapsed time from the computer's CPU starts processing the Jacobi algorithm and until the processing is completed, and compare it with the the elapsed CPU time for the eigvals(A) function from SciPy. The result is shown in table (2) for different matrix sizes $n \times n$, and reveals that the Jacobi algorithm is quite inefficient. The number of similar transformations the Jacobi's method need in order to diagonalize the matrix within the defined accuracy (subsequent similar transformations continues as long as the norm of the off diagonal elements in the transformed symmetric matrix $\mathbf{B}$ is off($\mathbf{B}$) $> 10^{-8}$) is also shown. From our results we see that when the number of discrete points n doubles, the number of similar transformations are rougly multiplied by four.

n	CPU Python	CPU Jacobi	Transformations N
10	3.1824E-04	1.0363E-02	1.5800E+02
20	1.9537E-04	1.4329E-01	6.7900E+02
40	4.1287E-04	2.1992E+00	2.8400E+03
80	4.4992E-03	3.4348E+01	1.1589E+04
160	1.0029E-02	5.4968E+02	4.7307E+04

Table 2: The elapsed CPU time for the SciPy function eigvals(A) and the Jacobi algorithm respectively as matrix size $n \times n$ of $\mathbf{A}$ in the buckling beam problem increases. The table also shows the number of similar transformations N in the Jacobi algorithm.

Problem 2:
We use Jacobi's method with $\rho_{max} = 5$ to find the eigenvalues of the system.

We compare our results of the four lowest eigenvalues, with the analytical solution given in the project 2 description: $\lambda = 3, 7, 11, 15, \cdots$. Notice that since the problem is discrete by its nature, it is a quantum mechanical problem, the eigenvalues are fixed. This is a major difference from the continous beam problem. For the eigenvalues we could calculate the electron's energy states for $l = 0$ by using the expressions given in the project description and find the corresponding eigenvectors for the scaled problem which gives the electron's scaled wave functions for $l = 0$. Table (3) shows the approximation of the four lowest eienvalues for an increasing matrix size $n \times n$. For the buckling beam problem, the eignevalues were accurate for a matrix size as small as $n \times n = 6 \times 6$. This is not the case here. The reason lies in the extra term $\rho^2 u(\rho)$ which is now included in the scaled equation, and which leads to the matrix elements d_i changing along the diagonal of the matrix.

n	λ_1	λ_2	λ_3	λ_4
10	2.9338	6.6598	10.142	13.345
20	2.9822	6.9102	10.780	14.592
40	2.9953	6.9767	10.943	14.900
80	2.9988	6.9940	10.986	14.979
160	2.9997	6.9985	10.997	14.999

Table 3: Accuracy increases for the four lowest eignevalues of the scaled one-electron potential well problem as matrix size $n \times n$ increases.

Problem 3:
We find the eignevalues for $\omega_r = 0.01, 0.25, 0.5, 1$ and 5 respectively, when the matrix size is $n \times n = 160 \times 160$. Table (4) shows the approximation of the four lowest eignevalues. We see that each of them increases as the frequency w_r increases. For $w_r = 0.25$, λ_1 coincides with the analytical form of the lowest scaled ground energy state calculated in [4].

ω_r	λ_1	λ_2	λ_3	λ_4
0.01	0.31163	0.68179	1.2228	1.9470
0.25	1.2499	2.1898	3.1498	4.1231
0.5	2.2298	4.1331	6.0705	8.0244
1.0	4.0566	7.9039	11.805	15.730
5.0	17.417	36.923	56.489	76.027

Table 4: Approximation of the four lowest eigenvalues of the scaled two-electron potential well problem for increasing frequencies ω_r. Matrix size is 160×160. For $w_r = 0.25$, λ_1 coincides with the analytical form of the lowest scaled ground energy state.

Conclusion

We have showed how Jacobi's method for finding the eigenvalues and the corresponding eigenvectors of a symmetric matrix works. We have programmed the eigenvalue part of it in Python and tested the algorithm on three physical eigenvalue problems, which is scaled in order to be without physical dimensions. When the equations were scaled, we were able to see beyond their different physical parameters and see the similarities in their mathematical expressions.

Physical problem 1 is a horizontal beam prevented from moving in vertical direction in its end points. A compressive force acting on one of its end points causes buckling of the beam. When tested against exact analytical eigenvalues, Jacobi's method showed great accuracy in calculating the eigenvalues in this problem for as few as 6 grid points (endpoints not included). In conjunction with this particular problem we also tested the time efficiency of the programmed Jacobi algorithm against the SciPy library's eigvals($\mathbf{A}$) function for increasing matrix sizes $n \times n$. The test revealed that our Jacobi algorithm is quite time inefficient. When the number of matrix rows and columns is doubled, the number of similar transfomations required to diagonalize it was multiplied by a factor of approximately four. When the matrix size reach $n \times n = 160 \times 160$ the Jacobi method starts to spend noticeable time in finding the eigenvalues. .

In physical problem 2, one electron moves in a three dimensional potential well, and in problem 3 two interacting electrons are moving in the well. The interaction between the two electrons is due to the Coulomb potential. In their radial directed and scaled equations, problem 2 and problem 3 has a potential term included, which separate their scaled equations from each other and from the scaled equation of the buckling beam. These potential terms cause the diagonal elements in the matrix of the matrix eigenvalue equation to vary along the diagonal. In order to converge towards the exact solutions of the eigenvalues, representing the ground energy states in the scaled equations, we therefore need a smaller step size h between the grid points, resulting in more grid points n. For matrix size $n \times n = 160 \times 160$ we get an accuracy within 2 to 3 leading digits after the decimal point, depending on the particular eigenvalue, compared to the exact, analytical eigenvalues in problem 2. In problem 3, we have one exact, analytical eigenvalue to compare our numerical results with. The matrix size $n \times n = 160 \times 160$ gives an accuracy of at least four leading digits after the decimal point. The results in problem 3 also shows that when the frequency w_r increases, the eigenvalues increases too.

Calculating the corresponding eigenvectors of the eigenvalues is not part of this project. In future work it is clear that the calculation of these eigenvectors should be included in the Jacobi algorithm as we have described in the Method section.

References

[1] Morten Hjort-Jensen: *Course material in Computational Physics.*
http://compphysics.github.io/ComputationalPhysics/doc/web/course

[2] Andreas Bjurstedt: *FYS4150-Computational Physics GitHub repository.*
https://github.com/abjurste/A19-FYS4150

[3] *SciPy.org*
https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.linalg.eigvals.html

[4] M. Taut: *Physical Review A, Volume48, page 3561, November 1993.*

Appendix: Computer programs

The program language used is Python. In Python, the first index in vectors and matrices is $i = 0$, not $i = 1$ which we used in the description of Jacobi's method in the methods section. Jacobi's method is programmed as a separate function in the Python script Jacobi_Eigen.py. The function is called upon when needed in the other scripts.

In the description of project 2 ,see [1], the remaining programming tasks are separated into four separate sub sections: Subsection 2b, 2c, 2d and 2e. These programming tasks are separated into four separate Python scripts:

- Project2b.py which deals with the buckling beam.
- Project2C_Unit_Tests.py which tests the Jacobi algorithm. Two so called unit tests are implemented. The first one tests if the algorithm is able to find the lagrest non diagonal element a_{ij} where $i < j$ in the symmertric matrix $\mathbf{A}$. The second one tests the Jacobi algorithm on a simple symmetric matrix in order to se if it can find the correct eigenvalues of the matrix. The algorithm passes the tests.
- Project2D_1electron.py which deals with the single electron moving in the potential well.
- Project2E_2electrons.py which deals with the problem when 2 electrons are moving in the potential well.

Jacobi_Eigen.py and Project2C_Unit_Tests.py are included below. For the other Python scripts, see [2].

Jacobi_Eigen.py:

```
import numpy as np
#import scipy.linalg as scl
from time import perf_counter




def Eigen_Jacobi(A):
    """
    The function Eigen_Jacobi(A) is diagonalizing the symmetric nxn matrix A using
```

```
Jacobi's method. The matrix eigenvalues then appear on the diagonalized matrix
diagonal. The function returns the diagonal matrix with the eigenvalues, the
number of times the matrix A had to be transformed by a rotation transformation
in order to be diagonalized, and the elapsed CPU time used from the start to
the end of diagonalizing process.
"""

a = np.shape(A)
n = a[0]

#Initial value of the norm of the non-diagonal elements in the symmetric
#matrix A n order to get in to while loop.
snond = 1.0
#Number of rotation transformations
st = 0

#Starting timer
start = perf_counter()

while snond > 1.0*10**(-8):  # and st < n**3:
    snond = 0.0
    Amax = 0.0
    #Number of rotation transformations increased by one more
    st = st+1

#Finding  the abs(maximum value) non-diagonal element in the symmetric matrix A
#and the norm of the non-diagonal elements in A.

    for i in range(n-1):
        for j in range(i+1,n):
            snond = snond + 2*A[i,j]**2
            if abs(A[i,j]) >= Amax:
                Amax = abs(A[i,j])
                k = i
                l = j
    snond = np.sqrt(snond)
#The rotation transfomation preserving the eigenvalues of matrix A
#First task: Calculating c = cos(theta) and s = sin(theta) for the orthogonal
#rotation matrix
```

```
    if A[k,l] != 0:
        tau = (1.0*A[l,l] - A[k,k])/(2*A[k,l])

        if tau > 0:
            t = -tau + np.sqrt(tau**2 + 1)
        else:
            t = -tau - np.sqrt(tau**2 + 1)

        c = 1.0/(np.sqrt(1 + t**2))
        s = t*c
    else:
        c = 1
        s = 0

#The rotation transfomation preserving the eigenvalues of matrix A
#Second task: Changing matrix elements with kk, ll, kl and lk indices

    a_kk = A[k,k]
    a_ll = A[l,l]
    A[k,k] = a_kk*c*c - 2*A[k,l]*c*s + a_ll*s*s
    A[l,l] = a_ll*c*c + 2*A[k,l]*c*s + a_kk*s*s
    A[k,l] = 0.0
    A[l,k] = 0.0


#The similarity transfomations preserving the eigenvalues of matrix A
#Third task: Changing matrix elements with ik, ki, il and li indices,
#where i!=k and i!=l

    for i in range(n):
        if i!=k and i!=l:
            a_ik = A[i,k]
            a_il = A[i,l]
            A[i,k] = a_ik*c - a_il*s
            A[i,l] = a_il*c + a_ik*s
            A[k,i] = A[i,k]
            A[l,i] = A[i,l]
    #Elapsed cpu time for finding the diagonal matrix.
    slutt = perf_counter()
    total = slutt - start
    #cpu_j = time.time() - c0




    return A, st, total


#For testing of the function: see Project 2C_Unit_tests.py
```

Project2C_Unit_Tests.py:

```
import numpy as np
import scipy.linalg as scl
from Jacobi_Eigen import *

#Test 1: An algortihm searching for the largest non-diagonal element
#in the symmetrical matrix A is neccesary in order to find the eigenvalues
#of this matrix with the Jacobi-method. This algorithm is part of the
#function "Jacobi_Eigen.py", and it is tested here.

#A symmetric test matrix A:
A = np.matrix([[1,2,3,4],[2,7,9,1],[3,9,6,3], [4,1,3,8]])
A=1.0*A
#Finding  the abs(maximum value) element in the symmetric matrix A

a = np.shape(A)
n = a[0]
Amax = 0.0

for i in range(n-1):
    for j in range(i+1,n):
        if abs(A[i,j]) >= Amax:
            Amax = abs(A[i,j])
            k = i
            l = j

print("""Largest matrix value: A[%d,%d] = A[%d,%d] = %d."""%(k,l,l,k,Amax))

#Test2: Checking that the function "Eigen_Jacobi(A)" finds the correct eigenvalues
#of the matrix A. The result is compared with the eigenvalues which the eigvals
#function from the Scipy libary calculates:


#Scipy function for calculating the eigenvalues of the symmetric matrix
eig_p = scl.eigvals(A)


#Finding the eigenvalues of the tridiagonal matrix by using Jacobi's method
[A,st,total] = Eigen_Jacobi(A)

#Sorting the numerical eigenvalues from lowest value up to highest value
eig_p = np.sort(eig_p)
a = np.diag(A)
a = np.sort(a)

print(""" """)
print("""Eigenvalues for the %d x %d symmetric matrix A."""%(n,n))
print("""   Python function   Jacobis method""")
for i in range(0,n):
    print("""  %15.8E   %15.8E """%(eig_p[i].real,a[i]))


#Running the script
"""
(base) M:\FYS4150-H19\Prosjekt2\Programmer>python Project2C_Unit_Tests.py
Largest matrix value: A[1,2] = A[2,1] = 9.

Eigenvalues for the 4 x 4 symmetric matrix A.
   Python function   Jacobis method
  -2.77703310E+00   -2.77703310E+00
  -1.06939477E+00   -1.06939477E+00
```

```
   8.06906286E+00    8.06906286E+00
   1.77773650E+01    1.77773650E+01
(base) M:\FYS4150-H19\Prosjekt2\Programmer>
"""
```