

FYS4150 - COMPUTATIONAL PHYSICS - PROJECT 2

EIMUND SMESTAD

EMAIL: eimundsm@fys.uio.no

2ND OCTOBER, 2014

Abstract

This report looks at different algorithms to solve eigenvalue problems for matrices and apply them on Schrödinger's equation for both a single electron and two electrons in a harmonic oscillator well with Coulomb repulsion between the electrons. I have investigated the Jacobi eigenvalue algorithm and Francis QR algorithm in more detail. I have done different optimization to be able to solve the eigenvalue problem faster, and compare the different algorithms. The results were a Jacobi algorithm that is 10 times faster than the standard implementation for solving the eigenvalue problems for the Schrödinger's equations in this report. Similarly I have made a QR algorithm for finding eigenvalues of symmetric tridiagonal matrix that solves the problem in this report significantly faster than `tqli` in `lib.cpp` of the course site, without loss of precision in the cases investigated in this report.

1 The radial Schrödinger's equation and numerical implementation

In this report I will study the radial part of Schrödinger's equation for both one and two electrons, which is as follows for a single electron

$$-\frac{\hbar^2}{2m} \left(\frac{1}{r^2} \frac{d}{dr} r^2 \frac{d}{dr} - \frac{\ell(\ell+1)}{r^2} \right) R(r) + V(r) R(r) = E R(r) ,$$

where $R(r)$ is the radial wave function, $r \in [0, \infty)$ is the radial parameter, ℓ is the quantum number for the orbital momentum, m is the mass, \hbar is the Planck's constant, $V(r)$ is the potential and E is the energy of the system. In this report I will only investigate the case $\ell = 0$, and we can then rewrite the Schrödinger's equation to this simplified form

$$-\frac{d^2}{d\rho^2} u(\rho) + V(\rho) u(\rho) = \lambda u(\rho) , \quad (1)$$

where $u(r) = rR(r)$, $r = \alpha\rho$, $\alpha^2 = -\frac{\hbar^2}{2m}$ and $\lambda = E$. This differential equation can be written on finite difference form

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

where subscript i is a indication of dependency to ρ_i , which is given by

$$\rho_i = \rho_{\min} + ih ,$$

and h is the step length

$$h = \frac{\rho_{\max} - \rho_{\min}}{n_{\text{step}}}$$

where n_{step} is the number of steps and $\rho \in [\rho_{\min}, \rho_{\max}]$. This difference equation can again be rewritten to a simplified form

$$e_{i+1}u_{i+1} + d_i u_i + e_{i-1}u_{i-1} = \lambda u_i, \quad (2)$$

where $e_i = -\frac{1}{h^2}$ and $d_i = \frac{2}{h^2} + V_i$. Introducing the Dirichlet boundary condition, $u_0 = u_{n_{\text{step}}} = 0$, this difference equation becomes a matrix eigenvalue problem with dimension $n_{\text{step}} - 1$, where we have a tridiagonal matrix.

1.1 Single electron in a harmonic oscillator well

The harmonic oscillator potential is given by

$$V(r) = \frac{1}{2}kr^2 \quad \text{with } k = m\omega^2,$$

where ω is the oscillator frequency. The possible energies of a harmonic oscillator system are

$$E_{n\ell} = \hbar\omega \left(2n + \ell + \frac{3}{2} \right) \quad \text{with } n, \ell \in \mathbb{N}_0, \quad (3)$$

where $\mathbb{N}_0 = \mathbb{N}_0^\infty$ is all the integers from zero to infinity. We can now write the Schrödinger's equation in (1) as

$$-\frac{d^2}{d\rho^2}u(\rho) + \rho^2 u(\rho) = \lambda u(\rho), \quad (4)$$

where we $\alpha = \left(\frac{\hbar^2}{mk}\right)^{\frac{1}{4}}$ and $\lambda = \frac{2m\alpha^2}{\hbar^2}E$ instead.

1.2 Two electrons in a harmonic oscillator well

The potential for Coulomb repulsion is given by

$$V(r) = \frac{\beta e^2}{r},$$

where r is the distance between the two electrons with $\beta e^2 = 1.44$ [eVnm]. From M. Taut's article [3] we have the following Shrödinger's equation relative between two electrons with Coulomb repulsion in a harmonic oscillator well where $\ell = 0$;

$$-\frac{d^2}{d\rho^2}u(\rho) + \omega_r^2 u(\rho) + \frac{1}{\rho} = \lambda u(\rho) \quad (5)$$

written on the form (1), where $\omega_r^2 = \frac{1}{4} \frac{mk}{\hbar^2} \alpha^4$, $\alpha = \frac{\hbar^2}{m\beta e^2}$, $\lambda = \frac{m\alpha^2}{\hbar^2} E_r$ and E_r is the relative energy between the two electrons. The solution have the eigenvalues

$$\lambda_n = 3 \left(\frac{\omega_r}{2} \right)^{\frac{2}{3}} + 2\sqrt{3}\omega_r \left(n + \frac{1}{2} \right) \quad \text{with } n \in \mathbb{N}_0, \quad (6)$$

with the ground state ($n = 0$) wave function

$$u_0(\rho) = \left(\frac{\sqrt{3}\omega_r}{\pi} \right)^{\frac{1}{4}} \exp \left(-\frac{\sqrt{3}}{2} \omega_r \left(\rho - (2\omega_r^2)^{-\frac{1}{3}} \right)^2 \right). \quad (7)$$

2 Eigenvalue algorithms

2.1 Diagonalize by matrix similarity

If we have a eigenvalue problem for a matrix \mathbf{A} with eigenvector \mathbf{x} and eigenvalue λ

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x},$$

we can transform it by a basis matrix \mathbf{S} to a similar matrix $\mathbf{B} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S}$ with the same eigenvalue λ but with a different eigenvector $\mathbf{S}^{-1}\mathbf{x}$;

$$\lambda\mathbf{S}^{-1}\mathbf{x} = \mathbf{S}^{-1}\mathbf{A}\mathbf{x} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S}\mathbf{S}^{-1}\mathbf{x} = \mathbf{B}\mathbf{S}^{-1}\mathbf{x}.$$

If we are able to diagonalize our matrix \mathbf{A} the eigenvalue problem becomes trivial to solve. However to directly diagonalize matrix \mathbf{A} of size $n \times n$ requires to solve a n -th polynomial problem, which is a very hard task. So we choose a easier way by iteratively zero out more and more non-diagonal elements of matrix \mathbf{A} , which will eventually give us the diagonal matrix

$$\mathbf{D} = \prod_{i=1}^n \mathbf{S}_{n-i}^{-1} \mathbf{A} \prod_{i=1}^n \mathbf{S}_i.$$

To make the problem simple to solve we define a change of basis matrix $\mathbf{S}_{i,j}$ with the elements

$$s_{ijkl} = \begin{cases} s_{ij} & \text{when } k = i \text{ and } \ell = j \\ 1 & \text{when } k = \ell \in \mathbb{N}_1^n / \{i, j\} \\ 0 & \text{elsewhere,} \end{cases} \quad (8)$$

where the strategy is for this change of basis matrix \mathbf{S} will make $b_{ij} = b_{ji} = 0$ when we do a matrix similarity transformation of \mathbf{A} to \mathbf{B} . The matrix elements of $\mathbf{S}_{i,j}^{-1}$ is then given by

$$s_{ijkl}^{-1} = \begin{cases} \frac{s_{jj}}{s_{ii}s_{jj} - s_{ij}s_{ji}} & \text{for } k = \ell = i \\ -\frac{s_{ij}}{s_{ii}s_{jj} - s_{ij}s_{ji}} & \text{for } k = i \text{ and } \ell = j \\ -\frac{s_{ji}}{s_{ii}s_{jj} - s_{ij}s_{ji}} & \text{for } k = j \text{ and } \ell = i \\ \frac{s_{ii}}{s_{ii}s_{jj} - s_{ij}s_{ji}} & \text{for } k = \ell = j \\ 1 & \text{for } k = \ell \in \mathbb{N}_1^n / \{i, j\} \\ 0 & \text{elsewhere.} \end{cases}$$

To see how this matrix similarity transformation behaves we must do the matrix multiplication $\mathbf{B} = \mathbf{S}_{i,j}^{-1} \mathbf{A} \mathbf{S}_{i,j}$ of the transformation;

$$b_{k\ell} = \sum_{m,o=1}^n s_{ijkm}^{-1} a_{mo} s_{ijol} = \begin{cases} s_{ii}^{-1} a_{ii} s_{ii} + s_{ij}^{-1} a_{ji} s_{ii} + s_{ii}^{-1} a_{ij} s_{ji} + s_{ij}^{-1} a_{jj} s_{ji} & \text{for } k = \ell = i \\ s_{ii}^{-1} a_{ii} s_{ij} + s_{ij}^{-1} a_{ji} s_{ij} + s_{ii}^{-1} a_{ij} s_{jj} + s_{ij}^{-1} a_{jj} s_{jj} & \text{for } k = i \text{ and } \ell = j \\ s_{ji}^{-1} a_{ii} s_{ii} + s_{jj}^{-1} a_{ji} s_{ii} + s_{ji}^{-1} a_{ij} s_{ji} + s_{jj}^{-1} a_{jj} s_{ji} & \text{for } k = j \text{ and } \ell = i \\ s_{ji}^{-1} a_{ii} s_{ij} + s_{jj}^{-1} a_{ji} s_{ij} + s_{ji}^{-1} a_{ij} s_{jj} + s_{jj}^{-1} a_{jj} s_{jj} & \text{for } k = \ell = j \\ s_{ki}^{-1} a_{i\ell} + s_{kj}^{-1} a_{j\ell} & \text{for } k \in \{i, j\} \text{ and } \ell \in \mathbb{N}_1^n / \{i, j\} \\ a_{ki} s_{i\ell} + a_{kj} s_{j\ell} & \text{for } \ell \in \{i, j\} \text{ and } k \in \mathbb{N}_1^n / \{i, j\} \\ a_{k\ell} & \text{elsewhere.} \end{cases} \quad (9)$$

What we want our matrix similarity transformation to do is to set $b_{ij} = b_{ji} = 0$, but we also need to ensure that the basis matrix \mathbf{S} for the transformation is invertible. Hence we have the following three equation we want to solve

$$s_{ii} s_{jj} - s_{ij} s_{ji} = 1 \quad (10)$$

$$a_{ji} s_{ij}^2 - (a_{ii} - a_{jj}) s_{jj} s_{ij} - a_{ij} s_{jj}^2 = 0 \quad (11)$$

$$a_{ij} s_{ji}^2 - (a_{jj} - a_{ii}) s_{ii} s_{ji} - a_{ji} s_{ii}^2 = 0. \quad (12)$$

If the matrix \mathbf{A} is indeed diagonalizable then the above three equations above solvable. We further observe that we have four unknowns in the three equations, which means that we can choose one of them as we want, I suggest $s_{ii} = 1$. I will not show the solution here, but it involves in principle in two second order equations $ax^2 + bx + c = 0$ which has a well known analytical solution $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Be aware of that the coefficient a may be zero which would make it a first order equations to solve instead. If the coefficient $b = 0$ in addition to $a = 0$, then the matrix \mathbf{A} is not diagonalizable.

2.2 Jacobi's eigenvalue algorithm

The Jacobi's eigenvalue algorithm is a special case of diagonalize by matrix similarity algorithm discussed in section 2.1, where I use a two dimensional rotation matrix \mathbf{R}_{ij} in n dimensional space which has the following elements

$$r_{ijk\ell} = \begin{cases} \cos \theta_{ij} & \text{when } k = i \text{ and } \ell = i \\ \sin \theta_{ij} & \text{when } k = i \text{ and } \ell = j \\ -\sin \theta_{ij} & \text{when } k = j \text{ and } \ell = i \\ \cos \theta_{ij} & \text{when } k = j \text{ and } \ell = j \\ 1 & \text{when } k = \ell \in \mathbb{N}_1^n / \mathbb{N}_j^{j+1} \\ 0 & \text{otherwise.} \end{cases} \quad (13)$$

which satisfies the basis matrix \mathbf{S} in (8). The rotation matrix can easily be shown to be a orthogonal matrix by realizing $\mathbf{R}_{ij}^{-1} = \mathbf{R}_{ij}^T$, because when we rotate an angle θ , we need to rotate an angle $-\theta$ to get back (or you could realize this by a more cumbersome way by actually do the matrix inversion).

By doing the matrix similarity transformation $\mathbf{B} = \mathbf{R}_{ij}^T \mathbf{A} \mathbf{R}_{ij}$, we get the following similar matrix elements according to (9),

$$b_{k\ell} = \begin{cases} a_{ii} \cos^2 \theta_{ij} + a_{jj} \sin^2 \theta_{ij} - (a_{ij} + a_{ji}) \sin \theta_{ij} \cos \theta_{ij} & \text{for } k = \ell = i \\ a_{ij} \cos^2 \theta_{ij} - a_{ji} \sin^2 \theta_{ij} + (a_{ii} - a_{jj}) \sin \theta_{ij} \cos \theta_{ij} & \text{for } k = i \text{ and } \ell = j \\ a_{ji} \cos^2 \theta_{ij} - a_{ij} \sin^2 \theta_{ij} + (a_{jj} - a_{ii}) \sin \theta_{ij} \cos \theta_{ij} & \text{for } k = j \text{ and } \ell = i \\ a_{jj} \cos^2 \theta_{ij} + a_{ii} \sin^2 \theta_{ij} + (a_{ij} + a_{ji}) \sin \theta_{ij} \cos \theta_{ij} & \text{for } k = \ell = j \\ a_{i\ell} \cos \theta_{ij} - a_{j\ell} \sin \theta_{ij} & \text{for } k = i \text{ and } \ell \in \mathbb{N}_1^n / \{i, j\} \\ a_{j\ell} \cos \theta_{ij} + a_{i\ell} \sin \theta_{ij} & \text{for } k = j \text{ and } \ell \in \mathbb{N}_1^n / \{i, j\} \\ a_{ki} \cos \theta_{ij} - a_{kj} \sin \theta_{ij} & \text{for } \ell = i \text{ and } k \in \mathbb{N}_1^n / \{i, j\} \\ a_{kj} \cos \theta_{ij} + a_{ki} \sin \theta_{ij} & \text{for } \ell = j \text{ and } k \in \mathbb{N}_1^n / \{i, j\} \\ a_{k\ell} & \text{elsewhere.} \end{cases}$$

The strategy is to set b_{ij} and b_{ji} to zero, which enables us by iteration to zero out every element in the matrix except the diagonal elements (as described in section 2.1). Unfortunately we have only one variable, namely θ_{ij} , which makes it impossible to diagonalize a general matrix \mathbf{A} by similarity. However when \mathbf{A} is a symmetric matrix, $a_{ij} = a_{ji}$, which gives us only one equation to solve to set both b_{ij} and b_{ji} to zero. For a symmetric matrix \mathbf{A} we have the following elements of the similarity matrix \mathbf{B}

$$b_{k\ell} = \begin{cases} a_{ii} \cos^2 \theta_{ij} + a_{jj} \sin^2 \theta_{ij} - 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} & \text{for } k = \ell = i \\ a_{ij} (\cos^2 \theta_{ij} - \sin^2 \theta_{ij}) + (a_{ii} - a_{jj}) \sin \theta_{ij} \cos \theta_{ij} & \text{for } k = i \text{ and } \ell = j, \text{ or } k = j \text{ and } \ell = i \\ a_{jj} \cos^2 \theta_{ij} + a_{ii} \sin^2 \theta_{ij} + 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} & \text{for } k = \ell = j \\ a_{i\ell} \cos \theta_{ij} - a_{j\ell} \sin \theta_{ij} & \text{for } k = i \text{ and } \ell \in \mathbb{N}_1^n / \{i, j\} \\ a_{j\ell} \cos \theta_{ij} + a_{i\ell} \sin \theta_{ij} & \text{for } k = j \text{ and } \ell \in \mathbb{N}_1^n / \{i, j\} \\ a_{ki} \cos \theta_{ij} - a_{kj} \sin \theta_{ij} & \text{for } \ell = i \text{ and } k \in \mathbb{N}_1^n / \{i, j\} \\ a_{kj} \cos \theta_{ij} + a_{ki} \sin \theta_{ij} & \text{for } \ell = j \text{ and } k \in \mathbb{N}_1^n / \{i, j\} \\ a_{k\ell} & \text{elsewhere.} \end{cases} \quad (14)$$

Now we are able to solve the following equation

$$b_{ij} = b_{ji} = a_{ij} (\cos^2 \theta_{ij} - \sin^2 \theta_{ij}) + (a_{ii} - a_{jj}) \sin \theta_{ij} \cos \theta_{ij} = 0,$$

by introducing $\tan \theta = \frac{\sin \theta}{\cos \theta}$ we can rewrite this equation to

$$\tan^2 \theta_{ij} + 2\tau_{ij} \tan \theta_{ij} - 1 = 0$$

where $\tau_{ij} = \frac{a_{jj} - a_{ii}}{2a_{ij}}$, which gives the solution
or

$$\tan \theta_{ij} = -\tau_{ij} \pm \sqrt{1 + \tau_{ij}^2} = \begin{cases} \frac{1}{\tau_{ij} - \sqrt{1 + \tau_{ij}^2}} & \text{when } \tau_{ij} < 0 \\ \frac{1}{\tau_{ij} + \sqrt{1 + \tau_{ij}^2}} & \text{otherwise,} \end{cases} \quad (15)$$

where I have chosen the \pm solution such that we avoid loss of numerical precision when subtracting almost equal parts. This makes $|\tan \theta| \leq 1$, because of the chose $\left| \tau_{ij} \pm \sqrt{1 + \tau_{ij}^2} \right| \geq 1$, which makes $\theta_{ij} \in \left[-\frac{\pi}{4} + n\pi, \frac{\pi}{4} + n\pi \right]$ where $n \in \mathbb{N}$. $\mathbb{N} = \mathbb{N}_{-\infty}^{\infty}$ is here all positive and negative integers including zero, which is usually indicated by \mathbb{Z} . However my notation \mathbb{N}_n^m is the integers between n and m , which may include negative integers, hence \mathbb{Z} is a redundant notation that is not needed. We know that $\tan \theta = \cot^{-1} \theta$ and we see from (15) that $-\tau_{ij} \pm \sqrt{1 + \tau_{ij}^2} = \left(-\tau_{ij} \pm \sqrt{1 + \tau_{ij}^2} \right)^{-1}$, which means that we can choose the following solution instead

$$\cot \theta_{ij} = -\tau_{ij} \mp \sqrt{1 + \tau_{ij}^2} = - \begin{cases} \frac{1}{\tau_{ij} - \sqrt{1 + \tau_{ij}^2}} & \text{when } \tau_{ij} < 0 \\ \frac{1}{\tau_{ij} + \sqrt{1 + \tau_{ij}^2}} & \text{otherwise,} \end{cases} \quad (16)$$

where we now have $\left| \tau_{ij} \pm \sqrt{1 + \tau_{ij}^2} \right| \geq 1$ which makes $\theta_{ij} \in \left[\frac{\pi}{4} + n\pi, \frac{3\pi}{4} + n\pi \right]$ instead. Now using the fact that $1 + \cot^2 \theta = \frac{1}{\sin^2 \theta}$ we can calculate

$$\sin \theta_{ij} = \frac{1}{\sqrt{1 + \cot^2 \theta_{ij}}}, \quad (17)$$

and $\cos \theta_{ij}$ is simply calculated $\cos \theta_{ij} = \sin \theta_{ij} \cot \theta_{ij}$. The reason for this change in solution of θ_{ij} is because we want $\sin^2 \theta_{ij}$ when we calculate the diagonal elements in (14) by rewriting them by using the trigonometrical relation $\cos^2 \theta + \sin^2 \theta = 1$;

$$\begin{aligned} b_{ii} &= a_{ii} \cos^2 \theta_{ij} + a_{jj} \sin^2 \theta_{ij} - 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} = a_{ii} (1 - \sin^2 \theta_{ij}) + a_{jj} \sin^2 \theta_{ij} - 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} \\ &= a_{ii} + \left[(a_{jj} - a_{ii}) \sin^2 \theta_{ij} - 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} \right] \end{aligned} \quad (18)$$

$$\begin{aligned} b_{jj} &= a_{jj} \cos^2 \theta_{ij} + a_{ii} \sin^2 \theta_{ij} + 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} = a_{jj} (1 - \sin^2 \theta_{ij}) + a_{ii} \sin^2 \theta_{ij} + 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} \\ &= a_{jj} - \left[(a_{jj} - a_{ii}) \sin^2 \theta_{ij} - 2a_{ij} \sin \theta_{ij} \cos \theta_{ij} \right] \end{aligned} \quad (19)$$

Where the point here is that we calculate $\sin^2 \theta_{ij}$ before $\sin \theta_{ij}$ in (17) to reduce number of FLOPS we need. Note that it's better to use $\sin^2 \theta$ instead of $\cos^2 \theta$, because when we look at

$$\begin{aligned} b_{ii} &= a_{jj} + \left[(a_{ii} - a_{jj}) \cos^2 \theta_{ij} - 2a_{ij} \cos \theta_{ij} \sin \theta_{ij} \right] \\ b_{jj} &= a_{ii} - \left[(a_{ii} - a_{jj}) \cos^2 \theta_{ij} - 2a_{ij} \cos \theta_{ij} \sin \theta_{ij} \right], \end{aligned}$$

we see that a_{jj} corresponds to b_{ii} and a_{ii} corresponds to b_{jj} , which means that we need to backup either a_{ii} or a_{jj} before we do the calculation, so we don't overwrite the value before we have used in the calculation. However for the $\sin^2 \theta_{ij}$ expressions we have that a_{ii} corresponds to b_{ii} and a_{jj} corresponds to b_{jj} , which means that we don't need to backup a_{ii} or a_{jj} , and we can use the program operator $+=$ directly on the diagonal elements.

We see from the terms $a_{xy} \cos \theta_{ij} \pm a_{zw} \sin \theta_{ij}$ in (14) that elements that was zero in \mathbf{A} isn't necessarily zero in \mathbf{B} . And we therefore seeks to minimize the off diagonal elements by searching for the largest of

diagonal elements and zero it out by the Jacobi method. This minimize the number of rotation transformation we need to do, however it requires $O(n^2)$ to search for the maximum off diagonal element.

2.3 Francis QR algorithm

We want to find the eigenvalues of a tridiagonal symmetric matrix \mathbf{A}_0 of $n \times n$ with QR decomposition on

$$\mathbf{A}_0 = \mathbf{S}_0 \mathbf{U}_0 ,$$

where \mathbf{S}_0 is the orthogonal matrix \mathbf{Q} and \mathbf{U}_0 is the upper triangle matrix \mathbf{R} in the QR decomposition. We can show that the similar matrix \mathbf{A}_1 to \mathbf{A}_0 is linked to the QR decomposition as follows

$$\mathbf{A}_1 = \mathbf{S}_0^T \mathbf{A}_0 \mathbf{S}_0 = \mathbf{S}_0^T \mathbf{S}_0 \mathbf{U}_0 \mathbf{S}_0 = \mathbf{U}_0 \mathbf{S}_0 ,$$

where I have used the orthogonal property of \mathbf{S}_0 such that $\mathbf{S}_0^T \mathbf{S}_0 = \mathbf{I}$ (\mathbf{I} is the identity martrix). Now imagine that we use QR decomposition on \mathbf{A}_1 and find the similarity matrix \mathbf{A}_2 in the same manner. We can now show by induction that this process for step $i + 1$ by using the QR decomposition on \mathbf{A}_i ;

$$\mathbf{A}_i = \mathbf{S}_i \mathbf{U}_i , \quad (20)$$

and we now find the similarity matrix \mathbf{A}_{i+1} to \mathbf{A}_i as follows

$$\mathbf{A}_{i+1} = \mathbf{S}_i^T \mathbf{A}_i \mathbf{S}_i = \mathbf{S}_i^T \mathbf{S}_i \mathbf{U}_i \mathbf{S}_i = \mathbf{U}_i \mathbf{S}_i . \quad (21)$$

We assume that \mathbf{A}_i is a symmetric tridiagonal matrix with the elements

$$a_{ik\ell} = \begin{cases} a_{ik(k+|\ell-k|)} & \text{when } \ell \in \mathbb{N}_{k-1}^{k+1} \\ 0 & \text{otherwise,} \end{cases}$$

which I will show by induction. To find the upper matrix we use successively rotation matrices \mathbf{R}_{ij} to eliminate the elements in the lower triangle part of the tridiagonal matrix \mathbf{A}_i

$$\mathbf{U}_i = \prod_{j=1}^{n-1} \mathbf{R}_{i(n-j)}^T \mathbf{A}_i , \quad (22)$$

where the rotation matrix \mathbf{R}_{ij} is defined as

$$r_{ijk\ell} = \begin{cases} \cos \theta_{ij} & \text{when } k, \ell \in \mathbb{N}_j^{j+1} \\ \sin \theta_{ij} & \text{when } k = \ell - 1 = j \\ -\sin \theta_{ij} & \text{when } k - 1 = \ell = j \\ 1 & \text{when } k, \ell \in \mathbb{N}_1^n / \mathbb{N}_j^{j+1} \\ 0 & \text{elsewhere.} \end{cases}$$

Note that the rotation matrix is orthogonal which means that the elements of \mathbf{R}_{ij}^T is given by $r_{ijk\ell}^T = r_{ij\ell k}$. I continue by defining the matrix

$$\mathbf{A}_{i(j+1)} = \begin{cases} \mathbf{A}_i & \text{when } j = 0 \\ \mathbf{R}_{ij}^T \mathbf{A}_{ij} & \text{when } j \in \mathbb{N}_1^{n-1}, \end{cases}$$

which means that we are going to show that $\mathbf{U}_i = \mathbf{A}_{in}$, where the matrix \mathbf{A}_{ij} has the elements $a_{ijk\ell}$. I start by doing the matrix multiplication $\mathbf{A}_{i2} = \mathbf{R}_{i1}^T \mathbf{A}_{i1}$;

$$\begin{aligned} a_{i2k\ell} &= \sum_{m=1}^n r_{i1km}^T a_{i1m\ell} = \sum_{m=-1}^1 r_{i1k(\ell+m)}^T a_{i1(\ell+m)\ell} = \begin{cases} \sum_{m=-1}^1 r_{i1k(\ell+m)}^T a_{i1(\ell+m)\ell} & \text{when } (k, \ell+m) \in {}^2\mathbb{N}_1^2 \\ a_{i1k\ell} & \text{otherwise.} \end{cases} \\ &= \begin{cases} \sum_{m=-1}^1 r_{i1k(\ell+m)}^T a_{i1(\ell+m)\ell} & \text{when } (k, \ell+m) \in {}^2\mathbb{N}_1^2 \text{ and } \ell \in \mathbb{N}_1^3 \\ a_{i1k\ell} & \text{otherwise.} \end{cases} \end{aligned}$$

The notation ${}^2\mathbb{N}_1^2 = \mathbb{N}_1^2 \times \mathbb{N}_1^2$, which means all the permutations of the natural number from 1 to 2; (1, 1), (1, 2), (2, 1) and (2, 2). We want our transformation to zero out the lower triangle element (2, 1)

$$a_{i221} = \sum_{m=-1}^1 r_{i12(m+1)}^T a_{i1(m+1)1} = \sum_{m=0}^1 r_{i12(m+1)}^T a_{i1(m+1)1} = 0,$$

which means that

$$a_{i2k\ell} = \begin{cases} \sum_{m=-1}^1 r_{i1k(\ell+m)}^T a_{i1(\ell+m)\ell} & \text{when } (k, \ell+m) \in {}^2\mathbb{N}_1^2, (k, \ell) \neq (2, 1) \text{ and } \ell \in \mathbb{N}_1^3 \\ 0 & \text{when } (k, \ell) = (2, 1) \\ a_{i1k\ell} & \text{otherwise.} \end{cases}$$

Note that all the lower off diagonal elements $a_{i2k\ell}$ is the same as $a_{i1k\ell}$, except $a_{i221} = 0$. Note also that a_{i213} is the only upper triangle element that isn't necessarily zero that was zero in matrix \mathbf{A}_{i1} . If we now continue the matrix multiplication to step j , $\mathbf{A}_{i(j+1)} = \mathbf{R}_{ij}^T \mathbf{A}_{ij}$;

$$\begin{aligned} a_{i(j+1)k\ell} &= \sum_{m=1}^n r_{ijkm}^T a_{ijm\ell} = \sum_{m=-1}^1 r_{ijk(\ell+m)}^T a_{ij(\ell+m)\ell} = \begin{cases} \sum_{m=-1}^1 r_{ijk(\ell+m)}^T a_{ij(\ell+m)\ell} & \text{when } (k, \ell+m) \in {}^2\mathbb{N}_j^{j+1} \\ a_{ijk\ell} & \text{otherwise.} \end{cases} \\ &= \begin{cases} \sum_{m=-1}^1 r_{ijk(\ell+m)}^T a_{ij(\ell+m)\ell} & \text{when } (k, \ell+m) \in {}^2\mathbb{N}_j^{j+1} \text{ and } \ell \in \mathbb{N}_{j-1}^{j+2} \\ a_{ijk\ell} & \text{otherwise.} \end{cases} \\ &= \begin{cases} \sum_{m=-1}^1 r_{ijk(\ell+m)}^T a_{ij(\ell+m)\ell} & \text{when } (k, \ell+m) \in {}^2\mathbb{N}_j^{j+1} \text{ and } \ell \in \mathbb{N}_j^{j+2} \\ a_{ijk\ell} & \text{otherwise,} \end{cases} \end{aligned}$$

where I have used that $a_{ijj(j-1)} = 0$ due to zero out of the previous rotation. We now want our transformation to zero out the lower triangle element $(j+1, j)$

$$a_{i(j+1)(j+1)j} = \sum_{m=-1}^1 r_{i,j,j+1,j+m}^T a_{ij(j+m)j} = \sum_{m=0}^1 r_{ij(j+1)(j+m)}^T a_{ij(j+m)j} = 0$$

where I have used the fact that $r_{ij(j+1)(j-1)}^T = 0$. This means that

$$a_{i(j+1)k\ell} = \begin{cases} \sum_{m=-1}^1 r_{ijk(\ell+m)}^T a_{ij(\ell+m)\ell} & \text{when } (k, \ell + m) \in {}^2\mathbb{N}_j^{j+1}, (k, \ell) \neq (j+1, j) \text{ and } \ell \in \mathbb{N}_j^{j+2} \\ 0 & \text{when } (k, \ell) = (j+1, j) \\ a_{ijk\ell} & \text{otherwise.} \end{cases} \quad (23)$$

I have now shown that all the lower off diagonal elements $a_{i(j+1)k\ell}$ is the same as $a_{ijk\ell}$, except $a_{i(j+1)(j+1)j} = 0$. Then using the assumption that \mathbf{A}_i is a tridiagonal matrix, induction gives us that $\mathbf{A}_{in} = \mathbf{U}_i$ is an upper triangle matrix. Furthermore $a_{i(j+1)j(j+2)}$ is the only upper triangle element that isn't zero that was zero in the matrix \mathbf{A}_{ij} , hence induction give us the following upper triangle matrix elements

$$u_{ik\ell} = \begin{cases} a_{ink\ell} & \text{when } \ell \in \mathbb{N}_k^{k+2} \\ 0 & \text{otherwise.} \end{cases} \quad (24)$$

Since $\mathbf{A}_i = \mathbf{S}_i \mathbf{U}_i$ and (22) means that

$$\mathbf{S}_i^T = \prod_{j=1}^{n-1} \mathbf{R}_{i(n-j)}^T,$$

and due to the similarity transformation $\mathbf{A}_{i+1} = \mathbf{S}_i^T \mathbf{A}_i \mathbf{S}_i = \prod_{j=1}^{n-1} \mathbf{R}_{i(n-j)}^T \mathbf{A}_i \prod_{i=1}^{n-1} \mathbf{R}_{ij}$ we have that

$$\mathbf{S}_i = \prod_{j=1}^{n-1} \mathbf{R}_{ij}.$$

To find the similarity matrix \mathbf{A}_{i+1} to \mathbf{A}_i we use successively rotation matrices \mathbf{R}_{ij} by defining the matrix

$$\mathbf{U}_{i(j+1)} = \begin{cases} \mathbf{U}_i & \text{when } j = 0 \\ \mathbf{U}_{ij} \mathbf{R}_{ij} & \text{when } j \in \mathbb{N}_1^{n-1}, \end{cases}$$

where $\mathbf{U}_{in} = \mathbf{A}_{i+1}$. Doing the matrix multiplication $\mathbf{U}_{i2} = \mathbf{U}_{i1} \mathbf{R}_{i1}$;

$$\begin{aligned} u_{i2k\ell} &= \sum_{m=1}^n u_{i1km} r_{i1m\ell} = \sum_{m=0}^2 u_{i1k(k+m)} r_{i1(k+m)\ell} = \begin{cases} \sum_{m=0}^2 u_{i1k(k+m)} r_{i1(k+m)\ell} & \text{when } (k+m, \ell) \in {}^2\mathbb{N}_1^2 \\ u_{i1k\ell} & \text{otherwise} \end{cases} \\ &= \begin{cases} \sum_{m=0}^1 u_{i1k(k+m)} r_{i1(k+m)\ell} & \text{when } (k+m, \ell) \in {}^2\mathbb{N}_1^2 \text{ and } k \in \mathbb{N}_1^2 \\ u_{i1k\ell} & \text{otherwise.} \end{cases} \end{aligned}$$

Note that all lower off diagonal elements $u_{i2k\ell}$ is the same as $u_{i1k\ell}$, except u_{i221} which now may be different than zero. If we now continue the matrix multiplication to step j , $\mathbf{U}_{i(j+1)} = \mathbf{U}_{ij} \mathbf{R}_{ij}$;

$$\begin{aligned}
u_{i(j+1)k\ell} &= \sum_{m=1}^n u_{ijkm} r_{ijm\ell} = \sum_{m=0}^2 u_{ijk(k+m)} r_{ij(k+m)\ell} = \begin{cases} \sum_{m=0}^2 u_{ijk(k+m)} r_{ij(k+m)\ell} & \text{when } (k+m, \ell) \in {}^2\mathbb{N}_j^{j+1} \\ u_{ijk\ell} & \text{otherwise} \end{cases} \\
&= \begin{cases} \sum_{m=0}^2 u_{ijk(k+m)} r_{ij(k+m)\ell} & \text{when } (k+m, \ell) \in {}^2\mathbb{N}_j^{j+1} \text{ and } k \in \mathbb{N}_{j-2}^{j+1} \\ u_{ijk\ell} & \text{otherwise,} \end{cases} \quad (25)
\end{aligned}$$

and we see that the lower off diagonal elements $u_{i(j+1)k\ell}$ is the same as $u_{ijk\ell}$, except $u_{i(j+1)(j+1)j}$ which now may be different than zero. Hence by induction I shown that the only non-zero lower off diagonal elements in $\mathbf{A}_{i+1} = \mathbf{U}_{in}$ is the elements $u_{ink(k-1)}$;

$$a_{(i+1)k\ell} = \begin{cases} u_{ink\ell} & \text{when } \ell \in \mathbb{N}_{k-1}^n \\ 0 & \text{otherwise.} \end{cases} \quad (26)$$

But we can actually show that \mathbf{A}_{i+1} is tridiagonal matrix by using the assumption that \mathbf{A}_i is a symmetric matrix. This we can do by showing that when \mathbf{A}_i is symmetric then \mathbf{A}_{i+1} is also symmetric by doing the matrix multiplication of the similarity transformation $\mathbf{A}_{i+1} = \mathbf{S}_i^T \mathbf{A}_i \mathbf{S}_i$ and using the fact that $a_{ik\ell} = a_{i\ell k}$;

$$a_{(i+1)k\ell} = \sum_{m,o=1}^n s_{ikm}^T a_{imo} s_{io\ell} = \sum_{m,o=1}^n s_{ikm}^T a_{iom} s_{io\ell} = \sum_{m,o=1}^n s_{io\ell} a_{iom} s_{ikm}^T = \sum_{m,o=1}^n s_{i\ell o}^T a_{iom} s_{imk} = a_{(i+1)\ell k}.$$

Since we have shown that \mathbf{A}_{i+1} is symmetric due to that \mathbf{A}_i is symmetric, and that \mathbf{A}_{i+1} has only non-zero elements in the lower off diagonal $a_{ik(k-1)}$, means that $a_{ik(k+1)}$ is the only higher off diagonal elements that are non-zero as well;

$$a_{(i+1)k\ell} = \begin{cases} a_{(i+1)k(k+|\ell-k|)} & \text{when } \ell \in \mathbb{N}_{k-1}^{k+1} \\ 0 & \text{otherwise.} \end{cases} \quad (27)$$

And we have indeed shown by induction that \mathbf{A}_i is symmetric tridiagonal matrix because the initial matrix \mathbf{A}_0 is symmetric tridiagonal matrix. Now if the off diagonal elements decreases for each QR transformation, then the similarity matrices \mathbf{A}_i will converge to a diagonal matrix, and hence we have found the eigenvalues.

Lastly I will show to calculate the rotation matrix \mathbf{R}_{ij} from the already established relation

$$a_{i(j+1)(j+1)j} = \sum_{m=0}^1 r_{ij(j+1)(j+m)}^T a_{ij(j+m)j} = a_{ij(j+1)j} \cos \theta_{ij} - a_{ijjj} \sin \theta_{ij} = 0,$$

which gives

$$\cos \theta_{ij} = \frac{a_{ijjj}}{\sqrt{a_{ijjj}^2 + a_{ij(j+1)j}^2}} \quad (28)$$

$$\sin \theta_{ij} = \frac{a_{ij(j+1)j}}{\sqrt{a_{ijjj}^2 + a_{ij(j+1)j}^2}}. \quad (29)$$

The method described here suggest to calculate the upper tridiagonal matrix \mathbf{U}_i and then similarity matrix \mathbf{A}_{i+1} , but this way of doing it would require we must store the calculated angles θ_{ij} . However from (23) we see that elements $a_{i(j+1)k\ell}$ with $k < j$ is not changed, and from (25) we see that elements $u_{i(j+1)k\ell}$ with $k > j + 1$, which means that we can calculate $u_{ij k\ell}$ after $a_{i(j+1)k\ell}$. This require us only to store one set of $\sin \theta_{ij}$ and $\cos \theta_{ij}$ values, because we need $\sin \theta_{ij}$ and $\cos \theta_{ij}$ after we have calculated $\sin \theta_{i(j+1)}$ and $\cos \theta_{i(j+1)}$, but for the steps afterwards we don't need the $\sin \theta_{ij}$ and $\cos \theta_{ij}$ anymore. Hence the only memory needed to solve the symmetric tridiagonal matrix eigenvalue problem is the memory of the symmetric tridiagonal matrix it self.

2.4 Eigenvectors

When we do a similarity transformation with the change of basis matrix \mathbf{S} , the eigenvector \mathbf{x} changes to $\mathbf{S}^{-1}\mathbf{x}$, as discussed in section 2.1. Which means that you would have to update your approximation to the eigenvectors for each similarity transform. Since an matrix of dimensionality n has n eigenvalues and therefore n eigenvectors, we would actually need to calculate new values for the eigenvectors n^2 times, which is very costly. Granted, for a simple rotation matrix there are only $2n$ elements we need to update. Never the less this has a huge impact on preformance on the Francis QR algorithm for instance, which has $n - 1$ rotations for each QR transformation.

However if we assume that we where able to calculate a very good approximation of the eigenvalues, we can calculate the eigenvectors afterwards. For symmetric tridiagonal matrix this becomes very simply. The idea is to use Gaussian elimination on (2) to eliminate all e_{i+1} elements (lower off diagonal elements). In our case of the radial Schrödinger equation, we can rewrite (2) such that the elements $e_i = -1$, which is very beneficial with regard to number of FLOPS needed. So I will continue with $e_i = -1$.

I start by using the Gaussian elimination to eliminate the element a_{21} in symmetric tridiagonal matrix \mathbf{A} of dimensionality n with $e_i = -1$. This gives

$$\check{a}_{22} = a_{22} - \lambda - \frac{1}{\check{a}_{11}},$$

where $\check{a}_{11} = a_{11}$ and λ is our approximated eigenvalue. And we can show by induction that

$$\check{a}_{ii} = a_{ii} - \lambda - \frac{1}{\check{a}_{(i-1)(i-1)}} \quad \text{for } i \in \mathbb{N}_2^{n-1} \text{ with } \check{a}_{11} = a_{11}. \quad (30)$$

If the λ is an good approximation to a eigenvalue to \mathbf{A} , we will then get that $(a_{nn} - \lambda)x_n - x_{n-1} \approx c(\check{a}_{(n-1)(n-1)}x_{n-1} - x_n)$, where c is a constant. By assuming that this approximation is good, we can exclude row n , and we can easily show that the eigenvector elements x_i are related as follows

$$x_{i+1} = \check{a}_{ii}x_i \quad \text{for } i \in \mathbb{N}_1^{n-1}. \quad (31)$$

As you can see, we can now quite easily iteratively calculate an approximate eigenvector after we have calculated the approximated eigenvalue, and the performance cost is very low. Note that one has to start with choosing an value for x_1 , where a natural chose would be $x_1 = 1$. And then you would have to normalize the eigenvector after all the elements are calculated, if you want it normalized.

The disadvantage with this method is that (31) is approximately correct, which means that \check{a}_{ij} has an error. By doing the calculation in accordance with (31), we accumulate more and more errors for each iterations. This means that the eigenvector elements x_i early in the iteration has a very good approximation to the real eigenvector, but will have a very bad approximation at the end. When you calculate the new eigenvector $\mathbf{S}^{-1}\mathbf{x}$ for each rotation instead, you have distrusted the error between the eigenvector elements, and therefore don't have this problem.

However in our case when we study the Schrödinger equation for a electron, we let $\rho_{\max} \rightarrow \text{infty}$, but we are only interested at the start of the interval of ρ . Which means we are free to cut off the end elements of the eigenvector and still get what we want, and then this Gaussian method of finding the eigenvector will give a good approximation. And indeed you can start your Gaussian method of iteration at an arbitrary element, to target a specific are of the eigenvector that you want a good approximation.

3 Numerical implementation

3.1 Matrices

Since the properties of a matrices is important for choosing the most effective method for solving a problem with regard to speed and memory usage, have I made partial specializations of a template class `template<MatrixType Type, class T> class Matrix`, where `MatrixType` it the partial specialization and `class T` specializes the matrix class to the type `T`.

Matrix partial specialization

```
enum class MatrixType {
    Square ,
    SquareT ,
    Symmetric ,
    Tridiagonal ,
    TridiagonalSymmetric ,
    Tridiagonal_m1_X_m1 ,
    Tridiagonal_m1_2_m1 ,
    Tridiagonal_m1_2_m1_6n ,
    Tridiagonal_m1_2_m1_4n ,
    LU_decomposition ,
    tqli
};
```

This enables enables us to use `MatrixType` to tell the compiler which type of matrix we want, and the compiler will then link up the solvers and functionality that is tailored for the desired matrix. In addition the program does not need to allocate more memory that are needed to represent the properties of the matrix, which will save a lot of memory for tridiagonal matrices for instance. This is all well and fine, but actually not needed, we could just name the different matrices classes differently and we would achieve the the same thing. But it's still nicer to call all the different matrix class for `Matrix`, and not `Matrix1`, `Matrix2`, ..., `Matrixn` for instance. However the true power of partial specialization comes to light when a function takes a matrix as input, but don't care what kind of matrix it is but only that it's matrix. An example of such a function is a function that print out the content of a matrix;

Printing a matrix

```
template<MatrixType Type, class T> void MatrixCout(Matrix<Type, T>& matrix)
{
```

```

        for(unsigned int i = 0; i < matrix.n; i++) {
            for(unsigned int j = 0; j < matrix.n; j++)
                cout << matrix(i,j) << "\t";
            cout << '\n';
        }
        cout << '\n';
    }

template <class M, class T> T& MatrixElements<M,T>::operator() (const
    unsigned int& row, const unsigned int& col) {
    return matrix[row][col];
}

template <class T> T& MatrixElements<Matrix<MatrixType::Symmetric, T>, T>::
    operator() (const unsigned int& row, const unsigned int& col) {
    return row < col ? matrix[col-row][row] : matrix[row-col][col];
}

template <class T> T& MatrixElements<Matrix<MatrixType::
    TridiagonalSymmetric, T>, T>::operator() (const unsigned int row, const
    unsigned int col) {
    if(row == col)
        return b[row];
    else if(row-1 == col)
        return a[col];
    else if(row == col-1)
        return a[row];
    other = 0;
    return other;
}

```

Furthermore you can use the partial specialization to tell the compiler how you want two different partial specialized class to interact. An example is if you want to take matrix multiplication of a tridiagonal matrix and a square matrix, you could let the compiler figure out how this should be linked up without you actually needing to know what kind of matrices you are multiplying. The compiler will always keep track of the matrix type, even if it's almost impossible for you to know it. This gives you a very high level feature on a low programming level, which makes it easy to optimize your code for speed and memory usage without having to do the hard work to link up optimized code, the compiler does it for you. The disadvantage is that it requires some fancy programming skills to write such partial specialized classes that tells how the compiler should link up the different cases. And if you do something wrong with the syntax, the compiler is usually less then helpful to tell you what is wrong. But when the difficult work of actually writing such classes the application of them becomes quite robust and easy to use in combination with C11 features `auto` and `decltype`, which enables you to hide the cumbersome and long names of the classes with the partial specialization. One should also keep in mind that such a code put a lot of work load on the compiler and the different possibilities of combination of partial specialization is written as code at compile time by the compiler, which makes your program substantially larger in size.

Sometimes you want a partial specialization of a function, but unfortunately this is not supported, and you would have to do a work around to achieve it. An example is a base class of `Matrix` that sets value on a diagonal in the matrix. Sometimes you want only to set the same value on the entire diagonal, and other times you want to set an array on the diagonal. Make such a function a template function that calls the partial specialized classes static function with the template argument, and then you have

actually partially specialized your function;

The base class `MatrixDiagonal<M, T>`

```
template<class M, class T> class MatrixDiagonal {
private: M* owner;
private: template<class P> class Type {
    public: static inline void Diagonal(M* owner, const int diagonal,
        const P value, const unsigned int n) {
        unsigned int d = abs(diagonal);
        owner->n = n;
        unsigned int nmax = n - d;
        if(diagonal > 0) {
            for(unsigned int i = 0; i < nmax; i++)
                owner->operator()(i,i+d) = value;
        } else {
            for(unsigned int i = 0; i < nmax; i++)
                owner->operator()(i+d,i) = value;
        }
    }
};

private: template<class P> class Type<P*> {
    public: static inline void Diagonal(M* owner, const int diagonal,
        const P* value, const unsigned int n) {
        unsigned int d = abs(diagonal);
        owner->n = n;
        unsigned int nmax = n - d;
        if(diagonal > 0) {
            for(unsigned int i = 0; i < nmax; i++)
                owner->operator()(i,i+d) = value[i];
        } else {
            for(unsigned int i = 0; i < nmax; i++)
                owner->operator()(i+d,i) = value[i];
        }
    }
};

protected: MatrixDiagonal(M* owner) : owner(owner) {
}

public: template<class P> inline void Diagonal(const int diagonal,
    const P value) {
    Type<P>::Diagonal(owner, diagonal, value, owner->n);
}

public: template<class P> inline void Diagonal(const int diagonal,
    const P value, const unsigned int n) {
    Type<P>::Diagonal(owner, diagonal, value, n);
}
};
```

And now you can call the "same" function `Diagonal` when you have only one value or an array of values.

Example of usage of partial specialized `MatrixDiagonal<M, T>::Diagonal`

```
double* d = new double[10];
auto matrix = Matrix<MatrixType::TridiagonalSymmetric, double>(10);
matrix.Diagonal(0, d);
matrix.Diagonal(1, -1);
```

3.2 Jacobi's eigenvalue algorithm

This algorithms solve only the eigenvalue problem, which means that the eigenvectors are not calculated here.

3.2.1 Jacobi lecture

This is mostly a direct implementation of the Jacobi algorithm given in the lecture notes [2, p. 218]. Note that this implementation uses a complete square matrix even though it solves for a symmetric matrix. This means that half of the memory is redundant.

```
template<class T> class Matrix<MatrixType::Square, T>::JacobiMethod
{
void JacobiMethod(T error, unsigned int &num /* */) {
    MatrixIndex I;
    T s, c, t, tau, a_kk, a_ll, a_ik, a_il;
    num = 0;
    while (error < MaxAbs(this->matrix, this->n, I)) {
        tau = (this->matrix[I.j][I.j] - this->matrix[I.i][I.i]) /
            (2*this->matrix[I.i][I.j]);
        if (tau > 0)
            t = 1.0/(tau + sqrt(1.0 + tau*tau));
        else
            t = -1.0/( -tau + sqrt(1.0 + tau*tau));
        c = 1/sqrt(1+t*t);
        s = c*t;

        a_kk = this->matrix[I.i][I.i];
        a_ll = this->matrix[I.j][I.j];
        // changing the matrix elements with indices k and l
        this->matrix[I.i][I.i] = c*c*a_kk - 2.0*c*s*this->matrix[I.
            i][I.j] + s*s*a_ll;
        this->matrix[I.j][I.j] = s*s*a_kk + 2.0*c*s*this->matrix[I.
            i][I.j] + c*c*a_ll;
        this->matrix[I.i][I.j] = 0.0; // hard-coding of the zeros
        this->matrix[I.j][I.i] = 0.0;

        // and then we change the remaining elements
        for (int i = 0; i < this->n; i++) {
            if ( i != I.i && i != I.j ) {
                a_ik = this->matrix[i][I.i];
                a_il = this->matrix[i][I.j];
                this->matrix[i][I.i] = c*a_ik + s*a_il;
                this->matrix[I.i][i] = this->matrix[i][I.i]
                    ];
                this->matrix[i][I.j] = c*a_il - s*a_ik;
                this->matrix[I.j][i] = this->matrix[i][I.j]
                    ];
            }
        }
        num++;
    }
}
```

And the search function for the maximum absolute value in the off diagonal elements.

Matrix<MatrixType::Square, T>::MaxAbs

```
T MaxAbs(T** matrix , unsigned int n, MatrixIndex& I) {
    T max = 0, abs1;
    for(unsigned int i = 0, j; i < n; i++) {
        for(j = i+1; j < n; j++) {
            abs1 = abs(matrix[i][j]);
            if(abs1 > max) {
                max = abs1;
                I.i = i;
                I.j = j;
            }
        }
    }
    return max;
}
```

3.2.2 My Jacobi algorithms

My implementation of Jacobi's algorithm uses the class `Matrix<MatrixType::Symmetric, T>` which uses the fact that we are solving a symmetric matrix, and therefore store only the upper triangle matrix in memory. The indexing of this matrix is done diagonal number `d` and row number `r` (`matrix[d][r]`), to better exploit the symmetry of the matrix.

Matrix indexing of `Matrix<MatrixType::Symmetric, T>`

```
T& operator() (const unsigned int& row, const unsigned int& col) {
    return row < col ? matrix[col-row][row] : matrix[row-col][col];
}
```

This is especially useful when we use it in combination with `tqli` in `lib.cpp`, where you want to send the main diagonal and the side diagonal of the tridiagonal matrix. By indexing by diagonal number we can easily obtain a pointer to the diagonals in the matrix, without doing remapping of memory.

I made a function that does rotation transformation in the Jacobi algorithm, which is basically the same as done in the lecture note [2], but with the optimization discussed in section 2.2.

Matrix<MatrixType::Symmetric, T>::Rotate

```
void Rotate(unsigned int i, unsigned int j, unsigned int& num) {
    T b = 2*this->matrix[j][i];
    this->matrix[j][i] = 0; // bij=bji=0

    j += i; // Convert j from diagonalnumber to column number

    // Calculate cos and sin
    T a = this->matrix[0][j] - this->matrix[0][i];
    T t = a/b;
    if(t > 0)
        t = (T)1/(t+sqrt(1+t*t)); // Calculate -cot !
    else
        t = (T)1/(t-sqrt(1+t*t)); // Calculate -cot !
    T s2 = (T)1/(1+t*t);
    T s = sqrt(s2);
    T c = t*s;
```



```

a = a*s2 + b*s*c;          // s*c = - sin*cos because of t=-cot
this->matrix[0][i] += a;    // bii eq(18)
this->matrix[0][j] -= a;    // bjj eq(19)

unsigned int k, l, n;
// Calculate bik and bjk, row < i and row < j
for(k = i, l = j, n = 0; n < i; k--, l--, n++) {
    a = this->matrix[k][n];
    b = this->matrix[l][n];
    this->matrix[k][n] = a*c - b*s;
    this->matrix[l][n] = b*c + a*s;
}

// row > i and row < j
for(k++, l--, n++; n < j; k++, l--, n++) {
    a = this->matrix[k][i];
    b = this->matrix[l][n];
    this->matrix[k][i] = a*c - b*s;
    this->matrix[l][n] = b*c + a*s;
}

// row > i and row > j
for(k++, l++, n = this->_n-j; l < n; k++, l++) {
    a = this->matrix[k][i];
    b = this->matrix[l][j];
    this->matrix[k][i] = a*c - b*s;
    this->matrix[l][j] = b*c + a*s;
}
num++; // To be removed
}

```

The last three for loops does the same, but the reason for separating them in such a way is due to the fact that we have a symmetric matrix, where only the upper triangle of the matrix is allocated in memory. So the three different for loops handles the indexing differently to make sure we operate in the correct element in the matrix.

3.2.2.1 Jacobi

This implementation does basically the same as the lecture implementation shown in section 3.2.1, where the algorithm searches for the maximum absolute value of the off diagonal elements and zero it out by rotation transformation in the Jacobi algorithm as shown in 3.2.2.

Matrix<MatrixType::Symmetric, T>::JacobiMethod

```

T* JacobiMethod(T error, unsigned int& num /**/) {
    MatrixIndex I;
    T** offdiagonal = &this->matrix[l];
    unsigned int offn = this->_n-1;
    num = 0;
    while(error < MaxAbs(offdiagonal, offn, I))
        Rotate(I.j, ++I.i, num);
    return this->matrix[0];
}

```

Matrix<MatrixType::Symmetric, T>::MaxAbs

```

T MaxAbs(T** matrix , unsigned int n, MatrixIndex& I) {
    T max = 0, abs1;
    for(unsigned int i = 0, j; n; i++, n--) {
        for(j = 0; j < n; j++) {
            abs1 = abs(matrix[i][j]);
            if(abs1 > max) {
                max = abs1;
                I.i = i;          // I.i is the diagonal
                                // number = col-row
                I.j = j;          // I.j is the row number
            }
        }
    }
    return max;
}

```

3.2.2.2 Jacobi FD

The standard Jacobi algorithm searches for the maximum absolute value off diagonal element, which is quite time expensive to use since it's goes as $O(n^2)$. But by studying the behavior of the off diagonal elements during the Jacobi algorithm a pattern emerges regarding the approximate position of the maximum diagonal elements for a symmetric tridiagonal matrix. Hence one can develop a Jacobi algorithm that doesn't do the search for maximum absolute value off diagonal elements, but rather iterate over the elements in a certain way. I will present different ways of doing this and compare the results from the different methods.

The first method I made is the Jacobi Forward Diagonal iteration, which first does the Jacobi transformation on all the elements in diagonal next to the main diagonal (symmetry diagonal) of the matrix. And then goes to the next diagonal and do the Jacobi transformation, and continues to the last diagonal of one element in the corner of the matrix is reached. Then the process starts from the beginning again, till all the off diagonal elements is less than a given value.

Matrix<MatrixType::Symmetric, T>::JacobiMethodFD

```

T* JacobiMethodFD(T error , unsigned int& num /* */) {
    bool run = true;
    num = 0;
    while(run) {
        run = false;
        // Run through the diagonals
        for(unsigned int d = 1, r, n = this->n-1; n; d++, n--) {
            for(r = 0; r < n; r++) { // Run through the
                rows
                if(error < abs(this->matrix[d][r])) {
                    run = true;
                    Rotate(r, d, num);
                }
            }
        }
    }
    return this->matrix[0];
}

```

3.2.2.3 Jacobi RD

This is the Jacobi Reverse Diagonal iteration, which is the same as Jacobi FD but the process is reversed.

Matrix<MatrixType::Symmetric, T>::JacobiMethodRD

```
T* JacobiMethodRD(T error, unsigned int& num /* */) {
    bool run = true;
    num = 0;
    while(run) {
        run = false;
        // Run through the diagonals
        for(unsigned int d = this->n-1, r, p = 1; d-->, p++) {
            for(r = 0; r < p; r++) { // Run through the rows
                if(error < abs(this->matrix[d][r])) {
                    run = true;
                    Rotate(r, d, num);
                }
            }
        }
    }
    return this->matrix[0];
}
```

3.2.2.4 Jacobi FC

This is the Jacobi Forward Column iteration, which starts at the lowest column number and does the Jacobi transformation on all it's elements before moving on to the next. Otherwise similar behavior to Jacobi FD.

Matrix<MatrixType::Symmetric, T>::JacobiMethodFC

```
T* JacobiMethodFC(T error, unsigned int& num /* */) {
    bool run;
    num = 0;
    do {
        run = false;
        for(unsigned int r = 0, d, n = this->n; r < this->n; r++, n--) {
            for(d = 1; d < n; d++) {
                if(error < abs(this->matrix[d][r])) {
                    run = true;
                    Rotate(r, d, num);
                }
            }
        }
    } while(run);
    return this->matrix[0];
}
```

3.2.2.5 Jacobi RC

This is the Jacobi Reverse Column iteration, which is the reverse of Jacobi FC.

Matrix<MatrixType::Symmetric, T>::JacobiMethodRC

```
T* JacobiMethodRC(T error, unsigned int& num /* */) {
    bool run;
    num = 0;
    do {
        run = false;
        for(unsigned int r = 0, d, n = this->n-1; r < this->n; r
            ++, n--) {
            for(d = n; d; d--) {
                if(error < abs(this->matrix[d][r])) {
                    run = true;
                    Rotate(r, d, num);
                }
            }
        }
    } while(run);
    return this->matrix[0];
}
```

3.2.2.6 Jacobi FR

This is the Jacobi Forward Row iteration, which starts at the lowest row number and does the Jacobi transformation on all it's elements before moving on to the next.

Matrix<MatrixType::Symmetric, T>::JacobiMethodFR

```
T* JacobiMethodFR(T error, unsigned int& num /* */) {
    bool run;
    num = 0;
    To be removed //
    do {
        run = false;
        for(unsigned int r, d = 1, c; d < this->n; d++) {
            for(r = 0, c = d; c; r++, c--) {
                if(error < abs(this->matrix[c][r])) {
                    run = true;
                    Rotate(r, c, num);
                }
            }
        }
    } while(run);
    return this->matrix[0];
}
```

3.2.2.7 Jacobi RR

This is the Jacobi Reverse Row iteration, which is the reverse of Jacobi FR.

Matrix<MatrixType::Symmetric, T>::JacobiMethodRR

```
T* JacobiMethodRR(T error, unsigned int& num /* */) {
    bool run;
    num = 0;
    do {
        run = false;
```

```

        for(unsigned int r, d = 1, c; d < this->n; d++) {
            for(r = d-1, c = 1; c <= d; r--, c++) {
                if(error < abs(this->matrix[c][r])) {
                    run = true;
                    Rotate(r, c, num);
                }
            }
        }
    } while(run);
    return this->matrix[0];
}

```

3.3 Francis QR algorithm

This algorithm is designed for solving a symmetric tridiagonal matrix, and is solved as described in section 2.3. Surprisingly this algorithm is quit slow, which is due to the fact that it needs $n_{\text{step}} - 1$ rotation transformation for each QR transformation, which you can see in Table 4.1.3.

However closer investigation showed that the QR algorithm gave vary big difference in absolute value of the off-diagonal elements. Many elements became small very fast, whereas others didn't. To speed up the algorithm I introduced a if check that determines whether the off-diagonal element is already smaller than a given value before doing the rotation transformation on it. If the off-diagonal element in question is smaller than the given value, it skips the rotation transformation. This reduces the number of rotation transformation that is needed to preform the QR algorithm significantly, rough estimates indicated a speed up of 25 times at tridiagonal matrix of dimensionality 100.

The first and last rotation in the QR transformation needs to be handled a little bit differently than the other rotation transformations, to be able to handle the corners of the matrix. Therefore the QR algorithm that I made has three section that does almost the same thing, but handles the three different stages of the QR transformation, namely first, intermediate and last rotation. This to avoid if checks that would be run frequently.

Matrix<MatrixType::TridiagonalSymmetric, T>::QRalgorithm

```

T* QRalgorithm(T error, unsigned int& num) {
    unsigned int i, j, n = this->n-2;
    T c, s, _c, _s, temp, temp2, max, a, b;

    num = 0;
    do {
        // First rotation in QR transformation
        max = 0;
        a = this->a[0];
        b = this->b[0];

        temp = sqrt((T)1/(a*a+b*b));    // Calculate sin and cos
        c = b*temp;
        s = a*temp;

        temp = a*s;                    // Eigenvalues
        this->b[0] = b*c*c + this->b[1]*s*s + 2*temp*c;
        this->b[1] = this->b[1]*c - temp;
    } while (max > error);
}

```

```

num++;

// The intermediate rotations in QR transformation
i = 1;
while(i < n) {
    if(abs(this->a[i]) > error) { // Offdiagonal
        element still to large?

        _c = c;
        _s = s;
        a = this->a[i];
        b = this->b[i];

        // Calculate sin and cos
        temp = (T)1/sqrt(a*a + b*b);
        c = b*temp;
        s = a*temp;

        temp = a*s; // Eigenvalues
        b = b*c + temp;
        this->a[i-1] = b*_s;
        temp2 = abs(this->a[i-1]);
        if(max < temp2)
            max = temp2;
        temp *= _c;
        this->b[i] = b*_c*c + temp*c + this->b[++i]
            ]*s*s;
        this->b[i] = this->b[i]*c - temp;

        num++;

    } else { // Offdiagonal element less then error
        if(s) { // If previous sin not zero , then
            update elements

            _c = c;
            _s = s;

            this->a[i-1] = this->b[i]*_s; //
            Eigenvalues
            this->b[i] = this->b[i]*_c;
            temp2 = abs(this->a[i-1]);
            if(max < temp2)
                max = temp2;

            c = 1;
            s = 0;

        }
        i++;
    }
}

// Last rotation in QR transformation
a = this->a[i];
b = this->b[i];

```

```

        temp = (T) 1/sqrt(a*a + b*b);    // Calculate sin and cos
        _c = b*temp;
        _s = a*temp;

        temp = a*_s;    // Eigenvalues
        b = b*_c + temp;
        this ->a[i-1] = b*s;
        temp2 = abs(this ->a[i-1]);
        if(max < temp2)
            max = temp2;
        temp *= c;
        this ->b[i] = b*_c*c + temp*_c + this ->b[++i]*_s*_s;
        b = this ->b[i]*_c - temp;
        this ->a[i-1] = b*_s;
        temp2 = abs(this ->a[i-1]);
        if(max < temp2)
            max = temp2;
        this ->b[i] = b*_c;

        num++;
    } while(max > error);

    return this ->b;
}

```

3.4 Eigenvectors

Here I will present the numerical implementation of finding the eigenvectors as discussed in section 5. This method is what I have used to calculate the eigenvectors together with the QR algorithm for finding the eigenvalues.

```

Matrix<MatrixType::Tridiagonal_m1_X_m1, T>::Eigenvector
T* Eigenvector(T value) {
    T temp = 1, temp2 = this ->b[0] - value;
    this ->b[0] = 1;
    for(unsigned int i = 1; i < this ->_n; i++) {
        temp *= temp2;
        temp2 = this ->b[i] - value - (T) 1/temp2;
        this ->b[i] = temp;
    }
    return this ->b;
}

```

This algorithm calculates only one eigenvector for a given eigenvalue.

4 Results

Order of relative error of v to u is calculated by

$$\epsilon = \log_{10} \left| \frac{v - u}{u} \right|.$$

4.1 Comparing different solvers

Number	tqli (lib.cpp)	QR	Jacobi (lecture)	Jacobi	Jacobi (FD)	Jacobi (RD)	Jacobi (FC)	Jacobi (RC)	Jacobi (FR)	Jacobi (RR)
10	6.8e-05	0.000125	0.000141	0.000114	8.4e-05	7.2e-05	0.000324	0.00011	0.000334	0.000106
100	0.010808	0.008582	0.373747	0.361826	0.048235	0.044294	0.482366	0.124545	0.483686	0.108885
1000	9.89616	2.66099	-	-	90.5831	80.3965	-	-	-	-
10000	19811.3	411.259	-	-	-	-	-	-	-	-

Table 4.1.1: Time used in seconds to solve the different eigenvalue and eigenvector solvers. The Jacobi solvers and QR algorithm terminates when off-diagonal elements are less than 10^{-6} . Used the case of single electron and $\rho_{\max} = 10$.

Number	QR	Jacobi (lecture)	Jacobi	Jacobi (FD)	Jacobi (RD)	Jacobi (FC)	Jacobi (RC)	Jacobi (FR)	Jacobi (RR)
10	-6.94463	-6.94463	-6.94463	-6.94463	-6.94462	-6.94445	-6.94462	-6.94445	-6.9446
100	-5.35675	-5.35046	-5.35104	-5.34988	-5.35022	-5.3323	-5.35485	-5.3323	-5.34152
1000	-2.79132	-	-	-3.229	-2.96929	-	-	-	-
10000	-0.64527	-	-	-	-	-	-	-	-

Table 4.1.2: Order of relative error to result of eigenvalues to the tqli in lib.cpp. The Jacobi solvers and QR algorithm terminates when off-diagonal elements are less than 10^{-6} . Used the case of single electron and $\rho_{\max} = 10$.

Number	QR	Jacobi (lecture)	Jacobi	Jacobi (FD)	Jacobi (RD)	Jacobi (FC)	Jacobi (RC)	Jacobi (FR)	Jacobi (RR)
10	857	129	128	324	269	1144	402	1144	384
100	115618	12830	12613	65886	61013	568584	165170	568584	147280
1000	13546473	-	-	10194528	8969921	-	-	-	-
10000	1384337487	-	-	-	-	-	-	-	-

Table 4.1.3: Number of rotations. The Jacobi solvers and QR algorithm terminates when off-diagonal elements are less than 10^{-6} . Used the case of single electron and $\rho_{\max} = 10$.

4.2 Single electron harmonic oscillator

$n_{\text{step}} \backslash \rho_{\text{max}}$	1	2	4	10	100	1000
4	0.697139	-0.220224	-0.768751	0.365163	2.51358	4.5149
10	0.821045	0.00785614	-1.36144	-0.801571	1.82361	3.83
20	0.843252	0.0435069	-2.1136	-1.06262	1.24455	3.26818
50	0.850155	0.0543602	-2.30935	-1.86932	0.341044	2.49633
100	0.851202	0.0559976	-2.18102	-2.46733	-0.881219	1.89877
1000	0.851547	0.056624	-2.14601	-4.47195	-2.45953	-0.895493
10000	0.865629	0.083662	-0.320132	-2.7285	-4.46025	-2.45874

Table 4.2.1: Order of relative error of the three lower eigenvalues with $\ell = 0$ for a single electron trapped in a harmonic oscillator well. The relative error is compared to the exact eigenvalues $\lambda_0 = 3$, $\lambda_1 = 7$ and $\lambda_2 = 11$. Solved with my QR algorithm with off-diagonal elements are less than 10^{-6} .

$n_{\text{step}} \backslash \rho_{\text{max}}$	1	2	4	10	100	1000
4	0.697139	-0.220224	-0.768751	0.365163	2.51358	4.5149
10	0.821045	0.00785616	-1.36144	-0.801571	1.82361	3.83
20	0.843252	0.043507	-2.11361	-1.06262	1.24455	3.26818
50	0.850155	0.0543607	-2.30944	-1.86933	0.341044	2.49633
100	0.851202	0.0559969	-2.18109	-2.46733	-0.881219	1.89877
1000	0.8515	0.0562747	-2.15945	-4.12992	-2.45956	-0.895493

Table 4.2.2: Order of relative error of the three lower eigenvalues with $\ell = 0$ for a single electron trapped in a harmonic oscillator well. The relative error is compared to the exact eigenvalues $\lambda_0 = 3$, $\lambda_1 = 7$ and $\lambda_2 = 11$. Solved with my tqli in lib.cpp

4.3 Two electrons harmonic oscillator with repulsive Coulomb interaction

$n_{\text{step}} \backslash \rho_{\text{max}}$	10	20	40	60	80	100	1000
10	0.290567	-0.513702	-2.33983	-3.26877	-2.3104	-1.85744	0.843536
100	0.293728	-0.507142	-2.13443	-2.16045	-2.16411	-2.16886	-1.85758
1000	0.293765	-0.507065	-2.13251	-2.15583	-2.15587	-2.15592	-2.1691

Table 4.3.1: Order of relative error of the ground state with $\ell = 0$ and $\omega_r = 0.01$ for two electrons trapped in a harmonic oscillator well with repulsive Coulomb interaction. The relative error is compared to the exact eigenvalues $\lambda_0 = 0.105041$. Solved with my QR algorithm with off-diagonal elements less than 10^{-6} .

$n_{\text{step}} \backslash \rho_{\text{max}}$	10	20	40	60	80	100	1000
10	-1.30549	-1.24813	-0.0891626	0.437514	0.74154	0.959596	3.00158
100	-1.07571	-1.08172	-1.10691	-1.15374	-1.23304	-1.37533	1.04043
1000	-1.07375	-1.07381	-1.07405	-1.07446	-1.07502	-1.07575	-1.38447

Table 4.3.2: Order of relative error of the ground state with $\ell = 0$ and $\omega_r = 0.5$ for two electrons trapped in a harmonic oscillator well with repulsive Coulomb interaction. The relative error is compared to the exact eigenvalues $\lambda_0 = 2.05658$. Solved with my QR algorithm with off-diagonal elements less than 10^{-6} .

$n_{\text{step}} \backslash \rho_{\text{max}}$	10	20	40	60	80	100	1000
10	-1.57751	-0.640193	0.442244	0.862346	1.13519	1.33955	3.35808
100	-0.922724	-0.932575	-0.975505	-1.0641	-1.26443	-2.23825	1.41663
1000	-0.919542	-0.919639	-0.92003	-0.920683	-0.9216	-0.922782	-2.43711

Table 4.3.3: Order of relative error of the ground state with $\ell = 0$ and $\omega_r = 1$ for two electrons trapped in a harmonic oscillator well with repulsive Coulomb interaction. The relative error is compared to the exact eigenvalues $\lambda_0 = 3.62193$. Solved with my QR algorithm with off-diagonal elements less than 10^{-6} .

$n_{\text{step}} \backslash \rho_{\text{max}}$	10	20	40	60	80	100	1000
10	-0.153129	0.690821	1.34894	1.71137	1.96484	2.16032	4.16326
100	-0.649211	-0.684726	-0.906906	-0.927603	-0.381454	-0.0261586	2.23493
1000	-0.638443	-0.638769	-0.640077	-0.642271	-0.645372	-0.649412	-0.0134

Table 4.3.4: Order of relative error of the ground state with $\ell = 0$ and $\omega_r = 5$ for two electrons trapped in a harmonic oscillator well with repulsive Coulomb interaction. The relative error is compared to the exact eigenvalues $\lambda_0 = 14.1863$. Solved with my QR algorithm with off-diagonal elements less than 10^{-6} .

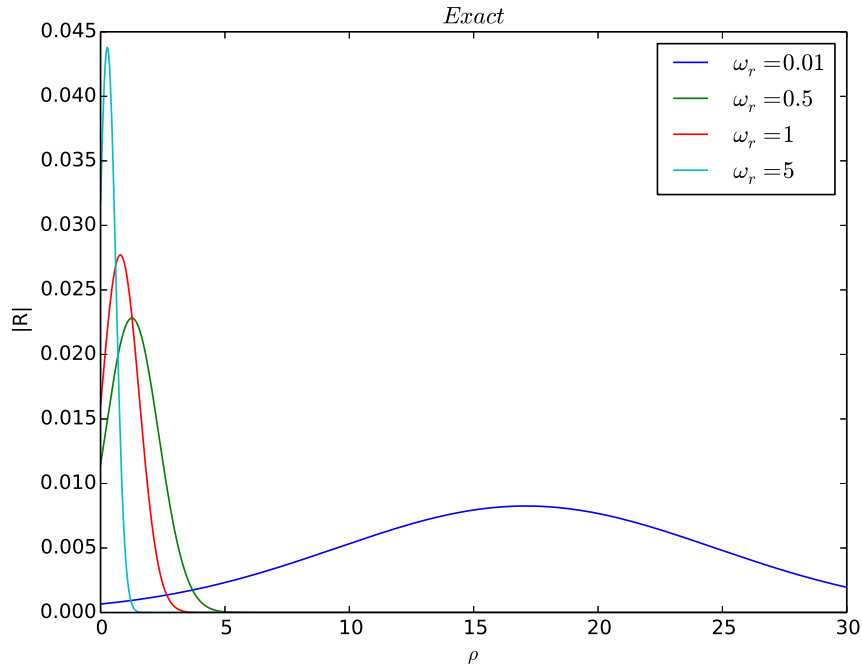


Figure 4.3.1: The normalize exact solution of the wave function for two electrons in a harmonic oscillator well with Coulomb repulsion (7).

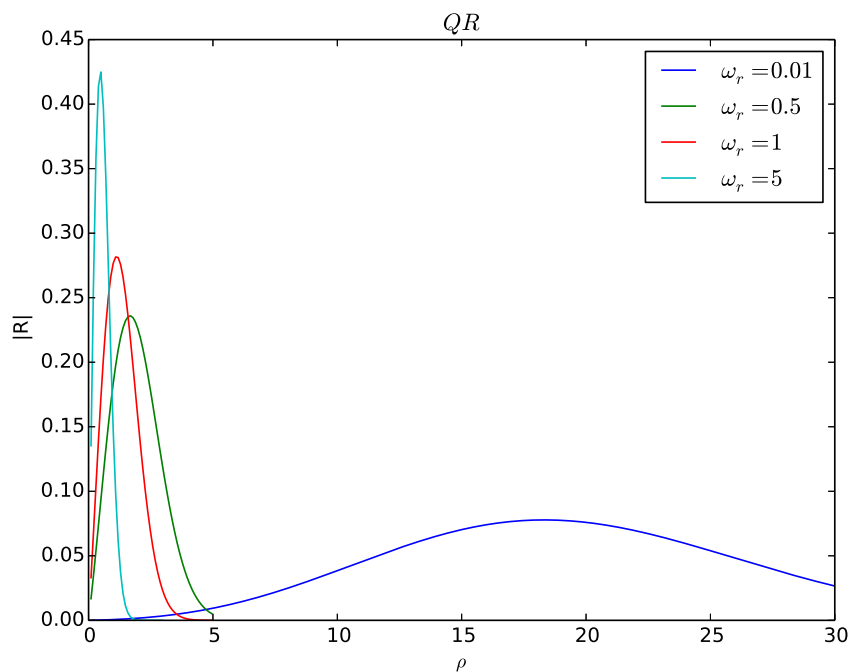


Figure 4.3.2: The normalize solution of the wave function from my own eigenvector algorithm for two electrons in a harmonic oscillator well with Coulomb repulsion, where Francis QR algorithm has been used to find the eigenvalues.

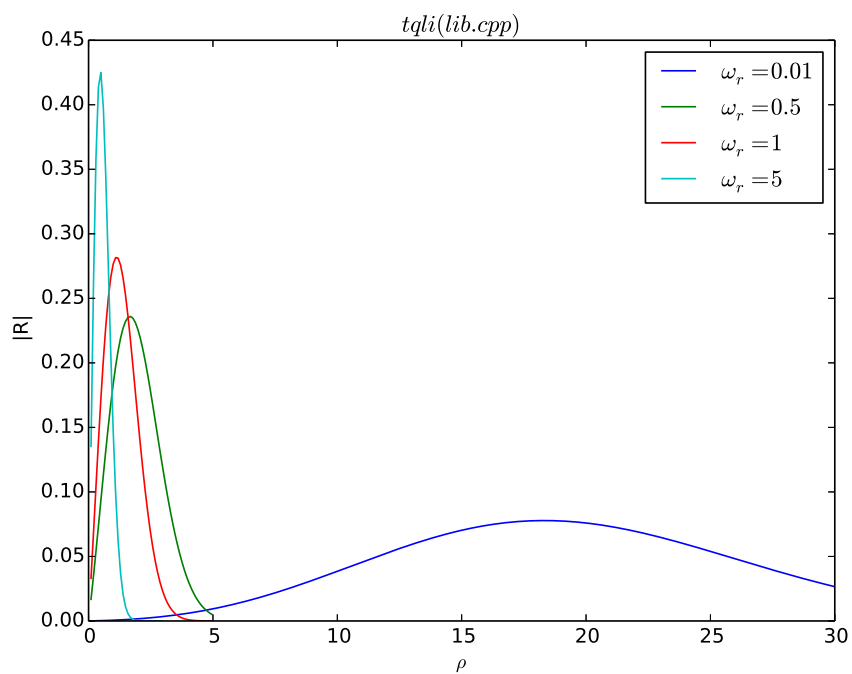


Figure 4.3.3: The normalize solution of the wave function from tqli in lib.cpp for two electrons in a harmonic oscillator well with Coulomb repulsion.

5 Conclusion

In this report I have shown that there are many different methods for solving a eigenvalue problem for a matrix, and the different solvers I have compared in section 4.1. Table 4.1.1 show big difference in computation time for different Jacobi solvers where we iterate over the off-diagonal elements in different ways. We see that both forward and reverse diagonal Jacobi method is very efficient to solve the eigenvalue problems in this report, which is 10 times faster than the standard Jacobi method where you search for the maximum absolute value off diagonal element. But Table 4.1.1 also show that certain iteration schemes for the Jacobi method is very inefficient, like the forward row and column iteration. All the solvers seems to go with $O(n^3)$ except the QR algorithm that seems to go with $O(n^2)$.

One should note that the `tqli` function calculates the eigenvectors as well, whereas the other solvers does not. This puts the `tqli` at a disadvantage in the speed comparison in Table 4.1.1, because the eigenvector calculation is quite expensive. However the implementation of the eigenvector algorithm that I made in that calculates the eigenvectors after the eigenvalue has been found, is very speed efficient and does not impact the results in Table 4.1.1. Though this eigenvector algorithm is not as numerical stable as the implementation in `tqli`, where you get a good approximation of the eigenvector at certain areas of the eigenvector. But taking care and guidelines as depicted in one can still get very good results with my eigenvector algorithm, which Figure 4.3.1, Figure 4.3.2 and Figure 4.3.3 shows. So my eigenvector algorithm with the QR algorithm that I made enables one to study heavy eigenvalue problems in matter of few minutes than rather of hours, which may be the difference between you getting insight to the problem or not, because you can run many more runs with a fast algorithm.

An unanswered question in this report is the results in Table 4.1.2 where all the solvers I made seems to diverge by the same amount relative to the `tqli` with higher numbers of steps. And decreasing the maximum absolute value of the off-diagonal elements did not seem to effect the results, which seems to indicate that the error is `tqli`. However Table 4.2.1, Table 4.2.2, Table 4.3.1, Table 4.3.2, Table 4.3.3 and Table 4.3.4 shows no significant difference in accuracy to known eigenvalues. The difference is that Table 4.1.2 compares all the eigenvalues and shows the largest difference, where as the other results shows only for few specific eigenvalues.

The algorithms that I made in this report terminates when the off-diagonal elements are less than a certain absolute value. This is however not an obvious indication of the accuracy of the results, and therefore a better way to input the desired accuracy of the solution would be beneficial.

6 Attachments

The files produced in working with this project can be found at <https://github.com/Eimund/UiO/tree/master/FYS4150/Project%202>

The source files developed are

1. [Matrix.h](#)
2. [project2.cpp](#)

7 Resources

1. QT Creator 5.3.1 with C11
2. Morten Hjorth-Jensen's `lib.cpp`
3. Eclipse Standard/SDK - Version: Luna Release (4.4.0) with PyDev for Python
4. Ubuntu 14.04.1 LTS
5. ThinkPad W540 P/N: 20BG0042MN with 32 GB RAM

References

- [1] Morten Hjorth-Jensen, *FYS4150 - Project 2 - Schrödinger's equation for two electrons in a three-dimensional harmonic oscillator well*, University of Oslo, 2014
- [2] Morten Hjorth-Jensen, *Computational Physics - Lecture Notes Fall 2014*, University of Oslo, 2014
- [3] M. Taut, *Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem*, Physical Review A Volume 48 Number 5, 1993
- [4] David J. Griffiths, *Introduction to Quantum Mechanics*, 2.ed, Pearson Education Inc., 2005
- [5] http://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors
- [6] http://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm
- [7] http://en.wikipedia.org/wiki/Matrix_similarity
- [8] http://en.wikipedia.org/wiki/Rotation_matrix
- [9] http://en.wikipedia.org/wiki/Orthogonal_matrix
- [10] http://en.wikipedia.org/wiki/Planck_constant
- [11] <http://en.wikipedia.org/wiki/Electron>
- [12] http://en.wikipedia.org/wiki/QR_algorithm
- [13] http://en.wikipedia.org/wiki/Gaussian_elimination