Project 2

FYS4150 Computational Physics

Even S. Håland

Abstract

In this project we see how the Schrödinger equation can be set up on discrete form as an eigenvalue problem. This is done for both one and two electrons, and we find that by scaling the equations appropriately the two problems look quite similar, except from a modification of the potential. The Jacobi algorithm is implemented and used to solve the eigenvalue problem. This algorithms is found to require $> N^2$ iterations, where N is the number of mesh points we consider, meaning that it take quite some time to solve the eigenvalue problem, at least if we want somewhat precise answers.

1 Introduction

In physics many problems and equations can be reduced to eigenvalue problems of the type $\mathbf{Ax} = \lambda\mathbf{x}$, where $\mathbf{A}$ is an $n \times n$ matrix. The characteristic polynomial of this equation is given by

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0,$$

and we realize that n shouldn't be very large before this problem becomes quite hard, if not impossible, to solve by hand. Therefore we need numerical methods and algorithms that can solve these kinds of problems for us. One such algorithm, which is the focus of this project, is the Jacobi algorithm (or Jacobi's method), which makes use of orthogonal transformations.

The particular eigenvalue problem we will consider is the Schrödinger equation, first for one electron in an harmonic oscillator potential, and then for two electrons. In the latter case we have to also consider the Coulomb repulsion between the electrons. The algorithm we develop will of course be used to solve these problems.

An important part of the project is to see how the equations in the two cases can be scaled to look quite similar. This makes life quite a lot easier for us when writing the programs, as we only have to write one program, and just change some input parameters corresponding to which problem we want to consider.

2 The Schrödinger equation

(This section follows very closely the theoretical introduction given in the project description, but for completeness sake I thought it would be nice to also include it in the report.)

2.1 One-particle case

We start by considering the radial part of the Schrödinger equation (SE) for one electron, which is given as

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

where (in our case) $V(r)$ is the harmonic oscillator (HO) potential

$$V(r) = \frac{1}{2}kr^2,$$

where $k = m\omega^2$. E is then the energy of the three dimensional HO, ω is the oscillator frequency, and these quantities are related by

$$E_{nl} = \hbar\omega\left(2n + l + \frac{3}{2}\right),$$

where $n = 0, 1, 2, \ldots$ and $l = 0, 1, 2, \ldots$. Throughout this project only cases with $l = 0$ will be considered.

By making the substitution $R(r) = (1/r)u(r)$ and introducing the dimensionless variable $\rho = (1/\alpha)r$ eq. (1) can be simplified to

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho) + \frac{k}{2}\alpha^2\rho^2 u(\rho) = Eu(\rho),$$

with boundary conditions $u(0) = u(\infty) = 0$. Further we can multiply the equation by $2m\alpha^2/\hbar^2$, and fix α so that

$$\frac{mk}{\hbar^2}\alpha^4 = 1.$$

If we also define

$$\lambda = \frac{2m\alpha^2}{\hbar^2}E$$

we end up with the eigenvalue equation

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

We must of course make a discrete approximation to the equation in order to solve the problem numerically. However, before moving on to that we should have a quick look at the changes that are introduced by adding a second electron to the problem.

2.2 Two-particle case

The radial SE for two electrons in an HO potential without any interactions is given by

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dr_1^2} - \frac{\hbar^2}{2m}\frac{d^2}{dr_2^2} + \frac{1}{2}kr_1^2 + \frac{1}{2}kr_2^2\right)u(r_1, r_2) = E^{(2)}u(r_1, r_2), \tag{3}$$

where $E^{(2)}$ is the two-electron energy. The solution to this equation is just the product of the wave functions for each electron. However, when we introduce the Coulomb interaction (which depends on the distance r between the electrons) eq. (3) is not very useful. For that reason we would like to introduce a new set of coordinates, namely the relative distance between the electrons,

$$\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2,$$

and the centre-of-mass coordinate for the system,

$$\mathbf{R} = \frac{1}{2}(\mathbf{r}_1 + \mathbf{r}_2).$$

The SE (still without interactions) can then be written as

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} - \frac{\hbar^2}{4m}\frac{d^2}{dR^2} + \frac{1}{4}kr^2 + kR^2\right)u(r, R) = E^{(2)}u(r, R),$$

where $r = |\mathbf{r}_1 - \mathbf{r}_2|$ and $R = \frac{1}{2}|\mathbf{r}_1 + \mathbf{r}_2|$. We then assume that the wave function is separable, so that $u(r, R) = \psi(r)\phi(R)$, and that the total energy is given by the sum of relative energy, E_r, and centre-of-mass energy, E_R, i.e.

$$E^{(2)} = E_r + E_R.$$

We can then add the term for the Coulomb interaction, which is given by

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

where $\beta e^2 = 1.44$ eVnm, and the r-dependent part of the SE becomes

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} + \frac{1}{4}kr^2 + \frac{\beta e^2}{r}\right)\psi(r) = E_r\psi(r).$$

As we did for the one-electron SE we now introduce the dimensionless variable $\rho = r/\alpha$, and scale the equation appropriately. After a few steps of manipulation we arrive to

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \omega_r^2\rho^2\psi(\rho) + \frac{1}{\rho} = \lambda\psi(\rho), \tag{4}$$

where

$$\omega_r^2 = \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4,$$

with α fixed so that

$$\frac{m\alpha\beta e^2}{\hbar^2} = 1,$$

and λ is defined as

$$\lambda = \frac{m\alpha^2}{\hbar^2}E.$$

It is noteworthy that the only differences between equations (2) and (4) is the factor ω_r^2 in the HO potential term, and the Coulomb repulsion term. This means that it is quite easy to make the transition between these cases when writing the code, which is in fact the main point of doing the scaling.

2.3 Discrete Schrödinger equation

As mentioned previously we must make a discrete approximation to the SE, and when doing so we will jump back to the one-electron problem, i.e. eq. (2).

The second derivative is approximated (up to $O(h^2)$) by

$$u'' \approx \frac{u(\rho + h) - 2u(\rho) + u(\rho - h)}{h^2},$$

where h is the step length. The minimum value of ρ is $\rho_{min} = \rho_0 = 0$, while the maximum value is in principle infinity. However, infinity is not a very practical "value" to work with, especially in a numerical context, which means that we must choose an appropriate maximum value $\rho_{max} = \rho_N$. The step length h is then defined as

$$h = \frac{\rho_N - \rho_0}{N},$$

where N is the number of mesh points we consider, and ρ is given as

$$\rho_i = \rho_0 + ih,$$

with $i = 1, 2, \ldots, N$. By using the short-hand notation $u(\rho_i + h) = u_{i+1}$, we can write the SE as

$$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + V_i u_i = \lambda u_i, \tag{5}$$

where $V_i = \rho_i^2$ is the HO potential. This is now an eigenvalue problem which can be compactly written as $\mathbf{Au} = \lambda\mathbf{u}$, where $\mathbf{A}$ is a matrix with diagonal elements

$$d_i = \frac{2}{h^2} + V_i$$

and non-diagonal elements

$$e_i = -\frac{1}{h^2}.$$

If we instead want to consider the two-electron case with the Coulomb interaction we simply change the potential V_i from ρ_i^2 to $\omega_r^2\rho_i^2 + 1/\rho_i$.

3 Jacobi's method

To summarize the situation we have now reduced the Schrödinger equation to an eigenvalue problem of the form $\mathbf{Au} = \lambda\mathbf{u}$, with $A \in \mathbb{R}^{N\times N}$. Our task is then to find the N eigenvectors $(\mathbf{u}_0, \mathbf{u}_2, \ldots, \mathbf{u}_{N-1})$ and eigenvalues $(\lambda_0, \lambda_2, \ldots, \lambda_{N-1})$ of A.

The idea behind Jacobi's method is to do a series of orthogonal transformations of the kind

$$(\mathbf{SAS}^T)(\mathbf{Sv}_i) = \mathbf{Sv}_i,$$

where $\mathbf{S}$ is an orthogonal matrix satisfying $\mathbf{SS}^T = \mathbf{I}$, and $\mathbf{v}_i$ is an orthogonal basis of $\mathbb{R}^N$. Our goal is then to eventually end up with a matrix on the left-hand side where all non-diagonal elements are (close to) zero, which means that the elements on the diagonal are (close to) the eigenvalues.

This method works because when we do an orthogonal transformation

$$\mathbf{B} = \mathbf{SAS}^T,$$

the eigenvalues of $\mathbf{B}$ are the same as those of $\mathbf{A}$, and it can be shown that [1] if $\mathbf{A}$ is real and symmetric (which it is in our case) there exists an orthogonal matrix, $\mathbf{M}$, such that

$$\mathbf{MAM}^T = diag(\lambda_0, \ldots, \lambda_{N-1}).$$

(A more careful discussion of this is found in for example refs. [1, 2].)

We can also see that if the basis vectors, $\mathbf{v}_i$, are orthogonal, that is[1]

$$\mathbf{v}_j^T \mathbf{v}_i = \delta_{ij},$$

then after an orthogonal transformation

$$\mathbf{w}_i = \mathbf{S}\mathbf{v}_i,$$

the dot product of the new vectors is

$$\begin{aligned} \mathbf{w}_j^T \mathbf{w}_i &= (\mathbf{S}\mathbf{v}_j)^T \mathbf{S}\mathbf{v}_i \\ &= \mathbf{v}_j^T \mathbf{S}^T \mathbf{S}\mathbf{v}_i \\ &= \mathbf{v}_j^T \mathbf{v}_i \\ &= \delta_{ij}, \end{aligned}$$

which means that orthogonality (and the dot product) is preserved by orthogonal transformations. This provides us with a nice way of testing our algorithm.

The next step is to choose a basis, $\mathbf{v}_i$, and a transformation matrix, $\mathbf{S}$. The basis vectors are chosen as simply as possible, namely

$$v_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad v_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \ldots, \quad v_{N-1} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix},$$

where each vector (of course) has n elements. The transformation matrix is chosen to be the matrix that rotates our system by an angle θ in a plane in the n-dimensional Euclidean space. Such a matrix has elements

$$s_{kk} = s_{ll} = \cos\theta, \ s_{kl} = -s_{lk} = -\sin\theta, \ s_{ii} = 1,$$

where $i \neq k, l$, and for a specific rotation k and l are fixed numbers. All other elements of $\mathbf{S}$ are zero. So when doing the transformation

$$\mathbf{B} = \mathbf{SAS}^T,$$

[1] This relation actually states that the $\mathbf{v}_i$'s are *orthonormal*, and not just orthogonal.

the matrix $\mathbf{B}$ gets the following elements:

$$\begin{aligned}
b_{ii} &= a_{ii},\ i \neq k, l \\
b_{ik} &= a_{ik}\cos\theta - a_{il}\sin\theta,\ i \neq k, l \\
b_{il} &= a_{il}\cos\theta + a_{ik}\sin\theta,\ i \neq k, l \\
b_{kk} &= a_{kk}\cos^2\theta - 2a_{kl}\cos\theta\sin\theta + a_{ll}\sin^2\theta \\
b_{ll} &= a_{ll}\cos^2\theta + 2a_{kl}\cos\theta\sin\theta + a_{kk}\sin^2\theta \\
b_{kl} &= b_{lk} = (a_{kk} - a_{ll})\cos\theta\sin\theta + a_{kl}(\cos^2\theta - \sin^2\theta)
\end{aligned}$$

Since we eventually want all the non-diagonal elements to be zeros (within some tolerance) we should chose the angle θ so that $b_{kl} = b_{lk} = 0$. From the expression for b_{kl} above we can get the second order equation

$$\tan^2\theta + 2\tan\theta\tau - 1 = 0,$$

where $\tau = (a_{ll} - a_{kk})/2a_{kl}$, which has the roots

$$\tan\theta = -\tau \pm \sqrt{1+\tau^2}.$$

By using the relations

$$\tan\theta = \frac{\sin\theta}{\cos\theta} \quad \text{and} \quad \cos^2\theta + \sin^2\theta = 1,$$

we find that

$$\cos\theta = \frac{1}{\sqrt{1+t^2}} \quad \text{and} \quad \sin\theta = \cos\theta\tan\theta,$$

which we then apply to our matrix.

For every transformation we do we want to reduce the off-diagonal norm, defined as

$$\text{off}(A) = \sqrt{\sum_i \sum_j |a_{ij}|^2}, \quad i \neq j,$$

such that

$$\text{off}(B) < \text{off}(A),$$

and that this norm in the end should be approximately zero. To reduce the off-diagonal norm as much as possible we start each iteration by picking out the largest off-diagonal element, and hence determine the indices k and l.

4 Programs and implementation

All code written for the project can be found in the following git-repository:

https://github.com/evensha/FYS4150/tree/master/Project2/Programs

The code is mainly written in C++, while some plotting is done with python. The script which are relevant (and will be discussed in the following) are

- `Jacobi_algorithm.cpp`
- `Project2.cpp`
- `Project2_plotting_1p.py`
- `Project2_plotting_2p.py`
- `RunProject.py`

In addition to these scripts the repository also contains a repository named "Output", where all the output from the programs are stored.

4.1 Implementing the Jacobi algorithm

The first program, called `Jacobi_algorithm.cpp`, contains the implementation of the Jacobi algorithm. (The implementation follows quite closely the examples given in refs. [2, 3].) The program itself consists of the three following functions which are declared in the header file `Jacobi_algorithm.h`:

- `offdiag`
- `Jacobi_rotation`
- `do_Jacobi`

As mentioned in the previous section we should start by finding the largest off-diagonal element of the matrix we are considering, which is done by the `offdiag`-function. This function takes the matrix $\mathbf{A}$ as input argument, along with the indices of the largest off-diagonal element, and the dimension of $\mathbf{A}$, and returns the largest off-diagonal element as a double.

When we have located the largest off-diagonal element we are ready to perform the Jacobi-rotation, which is done with the `Jacobi_rotation`-function. The first input argument is the matrix, $\mathbf{A}$, on which we want to perform the transformation. The second input argument is the matrix $\mathbf{R}$, which contains the basis vectors. Then follows the indices k and l, which we get from the `offdiag`-function, and the dimension n of the space we are working with. The first thing that is done is to calculate τ, $\tan\theta$, $\cos\theta$ and $\sin\theta$ according to the formulas given in the previous section, and then the updated elements of $\mathbf{A}$ and $\mathbf{R}$ are calculated.

The last function is called `do_Jacobi`, and also takes $\mathbf{A}$, $\mathbf{R}$ and n as input, as well as a vector in which the eigenvalues will be stored. First we initialize $\mathbf{R}$, and define the tolerance and maximum number of iterations we want the algorithm to do. The tolerance is the value that we want to get all off-diagonal elements of $\mathbf{A}$ below, while the maximum number of iterations is just so that the algorithm can't go on "forever". Then we find the initially largest off-diagonal element, and start a "while"-loop that goes on until all off-diagonal elements are below the tolerance (or we have reached the maximum number of iterations). When the iterations are finished the eigenvalues should be on the diagonal of $\mathbf{A}$.

4.2 Testing the algorithm and solving the Schrödinger equation

The program `Project2.cpp` contains the `main` program, as well as a function called `Jacobi_tests`. The latter function runs some simple tests on our implementation of the Jacobi algorithm, and is called without any input arguments. Three different tests are implemented:

- **Eigenvalues:** The algorithm is tested on a 2×2-matrix with eigenvalues 1 and 6, and we check that the algorithm actually gives these eigenvalues.
- **Maximum off-diagonal element:** We define a 5×5-matrix, and check that the `offdiag`-function gives the correct matrix element.
- **Dot product:** We check that the dot product is preserved, using the same 5×5-matrix as in the previous bullet test.

By calling the `Jacobi_tests`-function the program is aborted if one of the tests fails.

After having tested the algorithm we are ready to attack the problem that is subject of this project, namely the Schrödinger equation. This is implemented in the main program, which must be run with either three three or four input arguments. The first argument specifies the problem we want to consider; either one particle (1pHO), two particles with no interaction (2pNoInt) or two particles with Coulomb repulsion (2pCoulomb). The second argument is the number of mesh points (i.e. dimension of the space), n, we want to consider, and the third argument is our choice for ρ_{max}. If we run the program for two particles we should also give a fourth argument, which is ω_r.

When the necessary quantities are defined we specify the potential, V_i, according to which problem we consider, set up the matrix $\mathbf{A}$ and run the algorithm. The resulting eigenvalues and eigenvectors are then paired up in a "map", and eigenvalues are sorted in increasing order. Finally the eigenvectors corresponding to the three lowest eigenvalues are written to file.

Alternatively, if we don't want to use the Jacobi algorithm, the problem can also be solved with Armadillo [4] by using the `eig_sym` function. This is also implemented in the main program, and comparing results from Armadillo and Jacobi could serve as a nice test that our algorithm works as it should. Later we will also compare the CPU time of these two methods.

The eigenvectors are plotted with the python scripts `Project2_plotting_1p.py` (one-particle case) and `Project2_plotting_2p.py` (two-particle case). For the one particle case the eigenvectors corresponding to the three lowest eigenstates are plotted, while for the two-particle the eigenvectors with and without Coulomb interaction are plotted together for the lowest eigenstate only.

Finally there is also a very small python script called `RunProject.py`. As we are supposed to solve the two-particle problem for several different values of ω_r, it is nice with a script that runs the program for all the cases we want to consider, which is what this script does.

5 Results

5.1 One-particle case

Knowing that the three lowest eigenvalues should be 3, 7 and 11, the first task is to find appropriate values for ρ_{max} and n. We start by choosing $N = 100$ (because the algorithm is fast for this value), and look at how the eigenvalues behave for some values of ρ_{max}. The results of this study is given in Table 1, and we see that amongst the four test values the best result is obtained by using $\rho_{max} = 5.0$.

Table 1: The three lowest eigenvalues calculated for different choices of ρ_{max} using 100 mesh points.

ρ_{max}	Eigenvalues
1.0	$\lambda_0 = 9.96153$
	$\lambda_1 = 39.0155$
	$\lambda_2 = 87.3475$
5.0	$\lambda_0 = 2.99922$
	$\lambda_1 = 6.99609$
	$\lambda_2 = 10.9906$
10.0	$\lambda_0 = 2.99687$
	$\lambda_1 = 6.98434$
	$\lambda_2 = 10.9617$
20.0	$\lambda_0 = 2.98744$
	$\lambda_1 = 6.93692$
	$\lambda_2 = 10.8453$

The precision can be increased further by increasing the number of mesh points (i.e. the dimensions of $\mathbf{A}$), and with $N = 400$ we get the following for the three lowest eigenvalues:

$$\lambda_0 = 2.99995$$
$$\lambda_1 = 6.99976$$
$$\lambda_2 = 10.9996$$

However, when increasing the dimensions of the matrix we also increase quite drastically the number of transformations that are needed, and hence the CPU time of the program. In Table 2 the number of iterations and CPU time required for some values of N is listed, together with the CPU time required to solve the problem with Armadillo. The first thing to notice is that the Jacobi algorithm requires *a lot* of iterations. For the specific case we are looking at here ($\rho_{max} = 5$ and tolerance 10^{-10}) the number of iterations required is about $1.8N^2$. We also notice that the Armadillo solves the problem *much* faster, meaning that the Jacobi algorithm is not the most efficient algorithm for dealing with this particular problem.

Table 2: Number of iterations and CPU time required for different values of N (using $\rho_{max} = 5$), compared to CPU time using Armadillo.

N	Iterations	CPU time	CPU time (Armadillo)
100	17627	2.18 s	0.01 s
200	71168	26.9 s	0.04 s
400	287520	7.7 min	0.12 s

In Figure 1 the squared eigenvectors for the three lowest eigenstates are shown. In this figure we can clearly see why $\rho_{max} = 5$ gives good precision, as the wave functions have almost just reached zero, and we don't "waist" many of our mesh points in areas where the wave function is zero anyway. Correspondingly we can see why $\rho_{max} = 1$ is a horrible choice, as we then would force the wave functions to zero before they naturally fade out. (Remember that the boundary condition in ρ_{max} is $u(\rho_{max}) = 0$.) Notice however that it is not certain that $\rho_{max} = 5$ is a good choice if we also want to consider higher states, i.e λ_3, λ_4 etc.

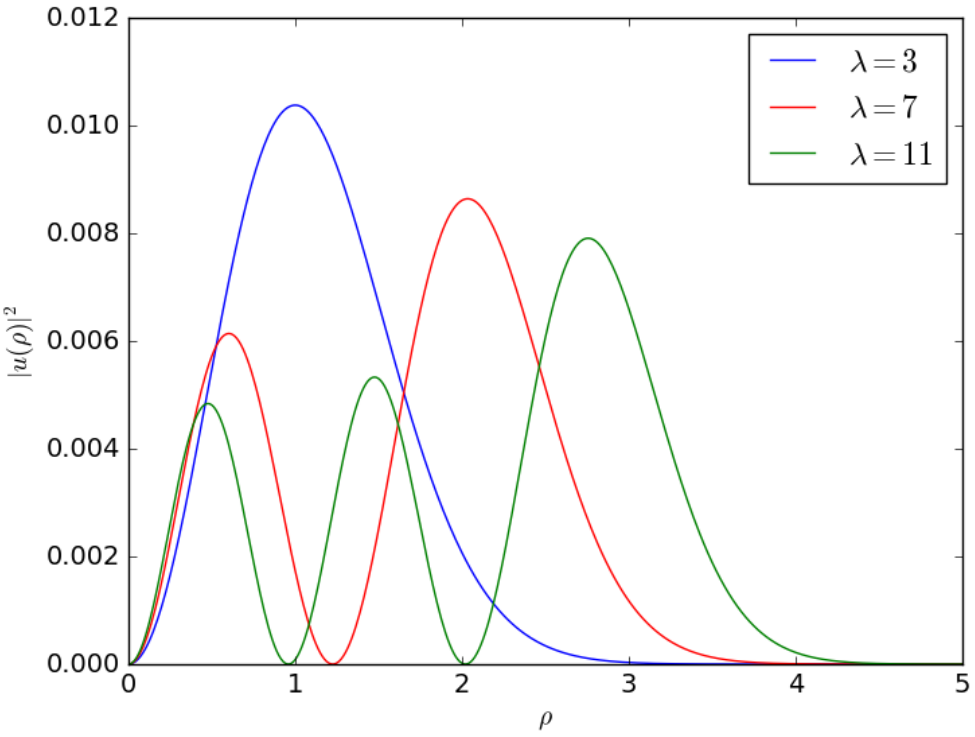

Figure 1: The eigenvectors corresponding to the three lowest eigenstates, calculated with $N = 400$ and $\rho_{max} = 5$.

5.2 Two-particle case

When considering only one electron we found that $\rho_{max} = 5$ was a good choice. When moving to the two-electron problem we need to reconsider this choice. The reason for this is that while the one-electron Schrödinger equation (after scaling) was independent of the harmonic oscillator frequency, the two-electron potential depends on ω_r (see eq. 4). This means that by varying ω_r we must also vary ρ_{max}. And by varying ρ_{max} we must also change N in order to obtain the same level of precision in our results. The choices made for these quantities (which are based on some "trying and failing") are listed in Table 3.

Table 3: Choice of ρ_{max} and N for the different values of ω_r.

ω_r	ρ_{max}	N
0.01	50	500
0.5	6	400
1	5	400
5	2	200

When running the program using the values from Table 3 we get the eigenvalues listed in Table 4. The first thing to notice is that for $\omega_r = 1$ and no interaction we get the same eigenvalue as for the one-electron case, which is to be expected, as the (scaled) equations look exactly the same. Further we see that eigenvalues for the interacting case are always larger than those for the non-interacting case. The eigenvalues are (proportional to) the relative energies of the systems, meaning that the interacting system has more energy than the non-interacting one, which seems natural. We also notice that the energy increases with ω_r, which we also would expect, as ω_r reflects the frequency of the harmonic oscillator.

Table 4: Eigenvalues for different values of ω_r for the ground state, both with and without Coulomb repulsion.

ω_r	λ_0 (No interaction)	λ_0 (Coulomb interaction)
0.01	0.0299997	0.105775
0.5	1.49998	2.23011
1	2.99995	4.05783
5	14.9992	17.4479

In Figure 2 the eigenvectors for the ground state are plotted for the different values of ω_r. We notice that the expected separation between the electrons is larger in the in the case with Coulomb interaction, which is quite natural as the Coulomb force is repulsive. We also notice that the difference in expected separation for the two cases becomes smaller when we increase ω_r, which is also to be expected, as the effect of the Coulomb repulsion should become less important when we increase the oscillator frequency and hence the energy of the electrons.

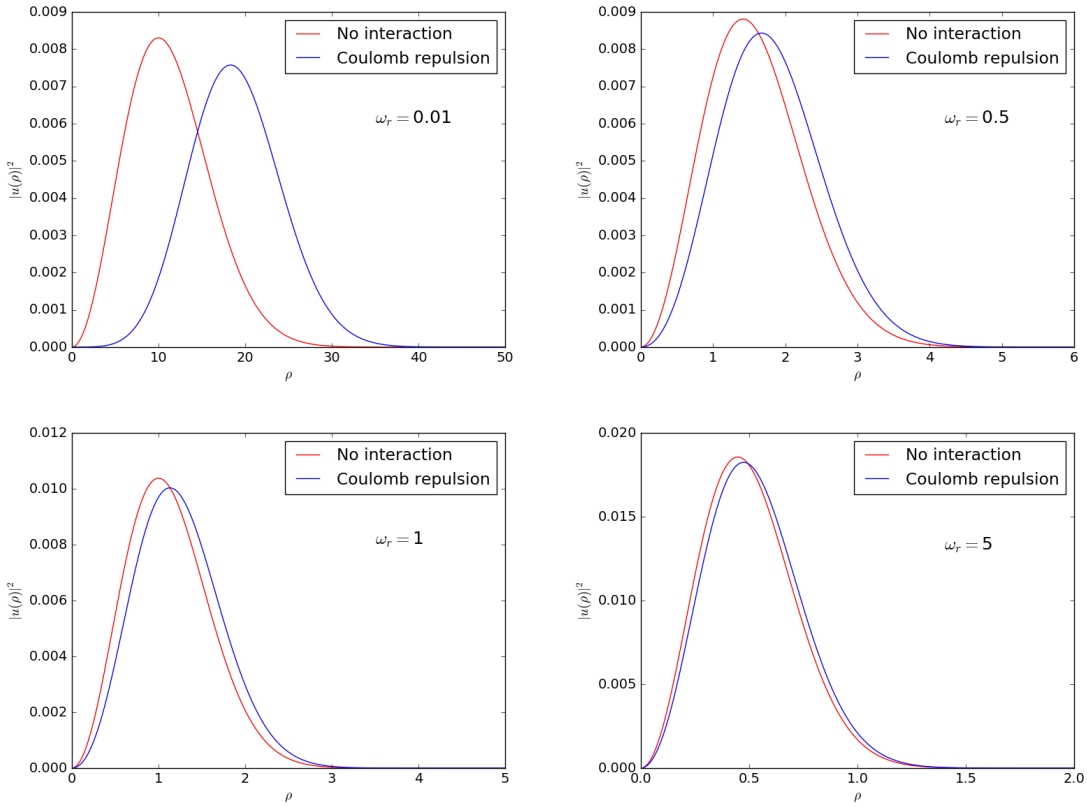

Figure 2: Eigenvectors for the ground states for the different values of ω_r.

6 Summary and conclusions

In this project we have implemented the Jacobi algorithm, and used it to solve the Schrödinger equation, both for one electron and for two electrons with Coulomb repulsion. We have seen how the equations can be scaled to take the same form. This gives the programs we wrote flexibility in terms of solving different kinds of eigenvalue problems, as the implementation is quite general.

We have also seen that the performance of our algorithm is quite poor. This is due to the fact that the Jacobi algorithm is very general, while the matrices we are considering are tridiagonal, which mean that applying the Jacobi algorithm can probably be considered as "overkill", and there are other methods (like Lanczos' algorithm) that could handled our problems more efficiently.

References

[1] G. Golub, C. Van Loan (1996), *Matrix Computations*, John Hopkins University Press.

[2] M. Hjort-Jensen (2015), *Computational Physics - Lecture Notes Fall 2015*, Department of Physics, University of Oslo.

https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf

[3] M. Hjort-Jensen (2017), *Computational Physics Lectures: Eigenvalue Problems*. http://compphysics.github.io/ComputationalPhysics/doc/pub/eigvalues/pdf/eigvalues-beamer.pdf

[4] C. Sanderson, R. Curtin (2016), Armadillo: a template-based C++ library for linear algebra, *Journal of Open Source Software*, Vol. 1, p. 26.