Project2 - FYS3150

Kristine Baluka Hein

Abstract

The aim of this report is to present a numerical method to solve for Schrödinger's equation for two electrons in three dimensions within a harmonic oscillator well, with and without Coulomb interaction. With some assumptions made, Schrödinger's equation can be rewritten into an eigenvalue problem. The method presented is Jacobi's rotation method.
We will first solve the equation for a single electron. It will then be possible to change our system for the single electron such that we can solve for two interacting electrons by just changing the oscillator potential. We will then see that the Jacobi's rotation method is not as effective as other methods, especially for large systems.

The project files can be found at **github**

1 Introduction

We wish to find a solution for Schrödinger's equation for two interacting electrons within an oscillator well. The solution will be found numerically, since we are not guaranteed that the Schrödinger's equation has precise analytical solutions which is straightforward to find. We will use Jacobi's iteration method since the idea behind the method is simple and is a decent example on the strength behind similarity transformations.

Here we will assume spherical symmetry so that the orbital angular momentum is zero and that the wave function u is stationary state. Schrödinger's equation will then become an eigenvalue problem. We will use spherical coordinates since spatial coordinates makes it more difficult in this case to solve for the wavefunction.
First we will limit our solver to solve for one non-interacting electron. When done so, it will be possible to extend our solver to solve for the interacting case just by changing the potential.
The method presented is Jacobi's rotation method. As the Jacobi's method is not the most efficient algorithm to find the eigenvalues and eigenvectors, the idea behind the method is fairly simple to understand.

2 Rewriting Schrödinger's equation

Before we can start looking at how to solve numerically the equation has to be rewritten in terms of less variables. It can be shown (see [3] page 1-2) that our equation reads:

$$-\frac{d^2}{d\rho^2}u(\rho)+\rho^2u(\rho)=\lambda u(\rho). \tag{1}$$

where $\rho=(1/\alpha)r$, $\alpha=\left(\frac{\hbar^2}{mk}\right)^{1/4}$, $\lambda=\frac{2m\alpha^2}{\hbar^2}E$, and the boundaries $u(0)=u(\infty)=0$.

Now it is possible to rewrite 1 as a discrete system by approximating the second derivative using the 3-point approximation.

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

Introduce $\rho_i = \rho_0 + ih$ where $h = \frac{\rho_n - \rho_0}{n-1}$ and $u_i = u(\rho_i)$ for $i = 0, \ldots, n-1$. Equation 1 will read:

$$\begin{aligned} -\frac{u(\rho_i + h) - 2u(\rho_i) + u(\rho_i - h)}{h^2} + \rho_i^2 u(\rho_i) &= \lambda u(\rho_i) \\ -\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + V_i u_i &= \lambda u_i \\ -u_{i+1}\frac{1}{h^2} + u_i\left(\frac{2}{h^2} + V_i\right) - u_{i-1}\frac{1}{h^2} &= \lambda u_i \end{aligned} \tag{2}$$

where $V_i = \rho_i^2$ is to be interpreted as the harmonic oscillator potential.
The problem with this discretization is that we cannot represent ∞. We have to choose an appropriate value for ρ_{n-1} which gives us reasonable results throughout the experiments.
We can rewrite equation 2 as a matrix equation. It will then become clear that we have to solve for an eigenvalue problem.
Let $d_i = \frac{2}{h^2} + V_i$ and $e_i = -\frac{1}{h^2}$. Then equation 2 reads

$$d_i u_i + e_{i-1} u_{i-1} + e_{i+1} u_{i+1} = \lambda u_i \tag{3}$$

Now, we can let d_i be the diagonal elements of our matrix, and e_i to be the offdiagonal elements of our matrix.
Let $\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-2} \end{pmatrix}$. Then the matrix equation of system 3 reads:

$$\begin{pmatrix} e_1 & d_1 & e_1 & 0 & \ldots & \ldots & 0 \\ 0 & e_2 & d_2 & e_2 & 0 & \ldots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & \ldots & 0 & e_{n-3} & d_{n-3} & e_{n-3} & 0 \\ 0 & \ldots & \ldots & 0 & e_{n-2} & d_{n-2} & e_{n-2} \end{pmatrix} \mathbf{u} = \lambda \mathbf{u} \tag{4}$$

The boundaries, u_0, u_{n-1} can be omitted as we already know which value they have. The eigenvectors can be interpreted as the energy eigenstates and the eigenvalues the energy levels of the electron we are looking at.
We have now our eigenvalue problem, 4. It is now possible to use our system in a solver which solves such eigenvalue problems.

3 Jacobi's rotation method

This report focuses on Jacobi's rotation iterative method. Jacobi's rotation method is not the most efficient way to find solve for eigenvalue problems compare to other method such as the QR algorithm. Suppose that we have an eigenvalue problem on the form

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

and wish to find the eigenvalue λ and the eigenvector $\mathbf{x}$.
We can then use Jacobi's rotation method to find the eigenvalues and the respective eigenvectors. The method rotates a given matrix A around the offdiagonal element $a_{k,l}$ which has the largest absolute value. This continues j times until the sum of the offdiagonal elements squared of $A^{(j)}$ becomes zero. For every rotation, the sum is reduced by $2|a_{k,l}^{(j-1)}|$ for every rotation (see [2], page 216-217 for proof). We assume that A is symmetric and quadratic.

3.1 The algorithm

We have the following algorithm from the method:

1) Check if we have to rotate

Find the largest off-diagonal element after rotation j,$j = 0, 1, \ldots,$ $|a_{k,l}^{(j)}|$ and check if it is greater than a given tolerance, ϵ. This is where the method presented differs slightly from the analytic method. In the analytic method we have to compute the sum of the diagonal elements and check if the sum is greater than the tolerance. However, as computing the sum for every iteration will be computationally demanding, we will limit our computation to just finding the largest absolute value of the off-diagonal elements of A.

2) The rotation

Now we rotate A around $|a_{k,l}^{((j-1))}|$. This is done by multiplying A with the transpose of a rotation matrix, R^T, on the left side and using that the rotation matrix R is orthogonal.
Our system will then read

$$R^T A^{(j-1)} R R^T \mathbf{x}^{(j-1)} = \lambda R^T \mathbf{x}^{(j-1)} \Rightarrow A^{(j)} R^T \mathbf{x}^{(j-1)} = \lambda R^T \mathbf{x}^{(j-1)} \Rightarrow A^{(j)} \mathbf{x}^{(j)} = \lambda \mathbf{x}^{(j)}$$

Doing the transformations enough times, we can expect to get a diagonal matrix. The elements of R are $R_{i,j} = \delta_{i,j}$ for $i \neq k, j \neq l$, $R_{k,k} = \cos\theta$, $R_{l,l} = \cos\theta$, $R_{k,l} = \sin\theta$ and $R_{l,k} = -\sin\theta$. Now we have to find the values for $\cos\theta$ and $\sin\theta$ before we apply the rotation.
After the rotation we have that

$$a_{kl}^{(j)} = (a_{kk}^{(j-1)} - a_{ll}^{(j-1)}) \cos\theta \sin\theta + a_{kl}^{(j-1)} (\cos^2\theta - \sin^2\theta) \tag{5}$$

We set $a_{kl}^{(j)} = 0$ to force the largest offdiagonal matrix element to not reappear in further rotations. After setting 5 to be zero, we get that

$$\tan^2\theta + 2\tau \tan\theta - 1 = 0 \Rightarrow \tan\theta = -\tau \pm \sqrt{1+\tau^2}$$

where $\tau = \dfrac{a_{ll}^{(j-1)} - a_{kk}^{(j-1)}}{2a_{kl}^{(j-1)}}$.

Note the numerical precision: We have to be aware that for large values of τ^2. If we look at $\sqrt{1+\tau^2}$, we will see that $\sqrt{1+\tau^2} \approx \sqrt{\tau^2}$ for large values of τ^2 due to truncation error of the numerical representation of out numbers. If, for example , we chose $\tan\theta = -\tau + \sqrt{1+\tau^2}$ when τ^2 is large and $\tau > 0$, we will have that $\tan\theta$ goes to zero. This implies $\theta = \pm k\pi$ for $k \in \mathbb{N}$. But if we look at the equation for $a_{kl}^{(j)}$, we will find that the only case for which $\theta = \pm k\pi$, is when $a_{kl} = 0$. But that is the case when we should be done rotating.
So we have to avoid the truncation error which gives us erroneous values for θ.
We should also be sure to have a value for $|\tan\theta| \leq 1$ such that the difference between the j-th and the (j-1)-th rotation goes to zero ([2], page 217). Let us consider this before handling the truncation error.
So $|\theta| \leq \frac{\pi}{4}$. This is done by some manipulation:

$$\begin{aligned}\tan\theta &= -\tau \pm \sqrt{1+\tau^2}\left(\frac{\tau \pm \sqrt{1+\tau^2}}{\tau \pm \sqrt{1+\tau^2}}\right) \\ &= \frac{1}{\tau \pm \sqrt{1+\tau^2}}\end{aligned}$$

We are now ensured that $|\tan\theta| \leq 1$.
For the truncation error, it is enough to set

$$\tan\theta = \begin{cases} \dfrac{1}{\tau + \sqrt{1+\tau^2}}, & \tau > 0 \\ \dfrac{1}{\tau - \sqrt{1+\tau^2}}, & \tau < 0 \end{cases}$$

To avoid potential cancellation in the divisor. And we are done considering the numerical problems around choosing reasonable value for $\tan\theta$.

Since we have an expression for $\tan\theta$, we can now compute $\cos\theta$ and $\sin\theta$.
In conclusion we have that

$$\begin{aligned} \tau &= \frac{a_{ll}^{(j-1)} - a_{kk}^{(j-1)}}{2a_{kl}^{(j-1)}} \\ \tan\theta &= \frac{1}{\tau \pm \sqrt{1+\tau^2}} \quad \textit{depending on } \tau \\ \cos\theta &= \frac{1}{\sqrt{1+\tan^2\theta}} \\ \sin\theta &= \cos\theta\tan\theta \end{aligned}$$

The elements of $A^{(j)}$, the resulting matrix after the j-th rotation, becomes:

$$\begin{aligned} a_{ii}^{(j)} &= a_{ii}^{(j-1)} && i \neq k, i \neq l \\ a_{ik}^{(j)} &= a_{ik}^{(j-1)}\cos\theta - a_{il}^{(j-1)}\sin\theta && i \neq k, i \neq l \\ a_{il}^{(j)} &= a_{il}^{(j-1)}\cos\theta + a_{ik}^{(j-1)}\sin\theta && i \neq k, i \neq l \\ a_{kk}^{(j)} &= a_{kk}^{(j-1)}\cos^2\theta - 2a_{kl}^{(j-1)}\sin\theta\cos\theta + a_{ll}^{(j-1)}\sin\theta \\ a_{ll}^{(j)} &= a_{ll}^{(j-1)}\cos^2\theta + 2a_{kl}^{(j-1)}\sin\theta\cos\theta + a_{kk}^{(j-1)}\sin\theta \\ a_{kl}^{(j)} &= (a_{kk}^{(j-1)} - a_{ll}^{(j-1)})\cos\theta\sin\theta + a_{kl}^{(j-1)}(\cos^2\theta - \sin^2\theta) \end{aligned}$$

The eigenvectors will also undergo a rotation.
They will be located in a matrix $S^{(j)}$. See 3.2 for a description of this property. So, for every rotation

$$\begin{aligned} s_{i,k}^{(j)} &= s_{i,k}^{(j-1)}\cos\theta - s_{i,l}^{(j-1)}\sin\theta \\ s_{i,l}^{(j)} &= s_{i,l}^{(j-1)}\cos\theta + s_{i,k}^{(j-1)}\sin\theta \end{aligned}$$

We will then continue rotating matrix $A^{(j)}$, and therefore $S^{(j)}$, until the tolerance is met as described in the first step.

The jacobi rotation method is implemented in `jacobi.cpp` in the function`jacobi_solver`:

```
while(find_max(A,ind,n) > tol)
{
   int k = ind[0], l = ind[1];
   double t = 0, c = 0,s = 0;

   double tau = (A[l][l] - A[k][k])/(2.*A[k][l]);
   t = (tau > 0 ? 1./(tau + sqrt(1. + tau*tau)) : 1./(tau - sqrt(1. + tau*tau)));
   c = 1./sqrt(1+t*t);
   s = t*c;

   double a_kk = A[k][k];
   double a_ll = A[l][l];
   double a_kl = A[k][l];
   A[k][k] = a_kk*c*c - 2*a_kl*s*c + a_ll*s*s;
   A[l][l] = a_ll*c*c + 2*a_kl*s*c + a_kk*s*s;
   A[k][l] = 0.;
   A[l][k] = 0.;
   for (size_t i = 0; i < n; i++)
   {
      if (i != k && i != l)
      {
         double a_ik = A[i][k];
         double a_il = A[i][l];
         A[i][k] = a_ik*c - a_il*s;
         A[i][l] = a_il*c + a_ik*s;
         A[k][i] = A[i][k];
         A[l][i] = A[i][l];
      }
   double r_ik = R[i][k];
   double r_il = R[i][l];
   R[i][k] = c*r_ik - s*r_il;
   R[i][l] = c*r_il + s*r_ik;
   }
}
```

where `R` is the matrix S.

3) After the rotations are done Now the eigenvalues for A can be found at the diagonal of $A^{(j)}$, and the corresponding eigenvectors in $S^{(j)}$. For the eigenvalue λ_j, the eigenvector is $\mathbf{s^{(j)}}_i$ which is the i-th column vector of $S^{(j)}$.

3.2 Properties

Since all of the successive rotations together form a orthogonal and real matrix S, and we have the property

$$S^T AS = \text{diag}(\lambda_0, \ldots, \lambda_{n-1})$$

with $\{\lambda_i\}_i$ being the eigenvalues, we have that, for column j of S, $\mathbf{s}_j$

$$A\mathbf{s}_j = \lambda_j \mathbf{s}_j$$

We will then have the eigenvectors stored in S, which is defined by the rotations done on $\mathcal{I}_n$, the identity matrix. We are guaranteed that the column vectors of $\mathcal{I}_n$ are eigenvectors of any matrix A since the vectors spans the whole vector space of A.

The rotation matrix is orthogonal, which means that it preserves the inner product of the vectors transformed. We would expect the eigenvectors to be orthogonal and normalized after the successive transformations, since the n eigenvectors of A (the column vectors of the identity matrix) are also orthogonal and normalized. To see why the inner product is preserved, let U be an orthogonal matrix and $\mathbf{y}$ be the resulting vector after the transformation of $\mathbf{x}$. Since U is orthogonal, we have that $U^T U = UU^T = \mathcal{I}_n$.
The inner product becomes $\mathbf{y}^T\mathbf{y} = (U\mathbf{x})^T(U\mathbf{x}) = \mathbf{x}^T U^T U\mathbf{x} = \mathbf{x}^T\mathbf{x}$ which shows that that the inner product is preserved throughout the transformation.

4 Non-interacting case

We can now solve 4 using Jacobi's rotation method for a single electron. In this case, the non-interacting case, we have the potential $V_i = \rho_i^2$.

4.1 Choosing the correct parameters

We have to make a choice for which number of grid points, n, and which ρ_{n-1} gives sufficient results since those values affects the precision of our computed results. We have that the lowest eigenvalues are $\lambda_0 = 3, \lambda_1 = 7, \lambda_2 = 11$ (see [3], page 2). We can then use our solver and check if we find a value for n and ρ_{n-1} which gives sufficient results, see Appendix A.

4.2 Time comparison, Jacobi's method with another method

Since the Jacobi rotation method has low convergence rate, we would also expect the time executing `jacobi_solver` in `jacobi.cpp` to be remarkable slower than any other used algorithm for find eigenvalues and the corresponding eigenvectors. Here we have used `numpy.linalg.eig` to compare with the values since `numpy.linalg.eig` is generally an efficient method.
The results after the time calculation of both methods can be found in Appendix B.
Through multiple runs of our solver, it is possible to see that the number of iterations are approximately $2n^2$(at least not greater than). Here we have made `jacobi_solver.cpp` to print out the number of iterations for $n = 75, 175, 300, 400, 650$ and $\rho_{n-1} = 5$:

n	number of iterations	$2n^2$
75	8698	11250
175	49774	61250
300	149081	180000
400	266160	320000
650	710972	845000

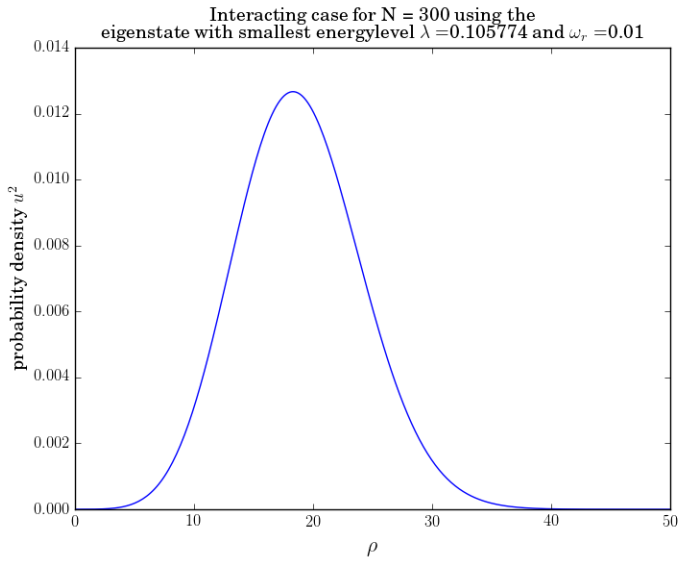
This shows that for very large systems, we should consider other methods to solve for the eigenvalue problem. This also explains why Jacobi's method is considerably slower than `numpy.linalg.eig` for larger systems.

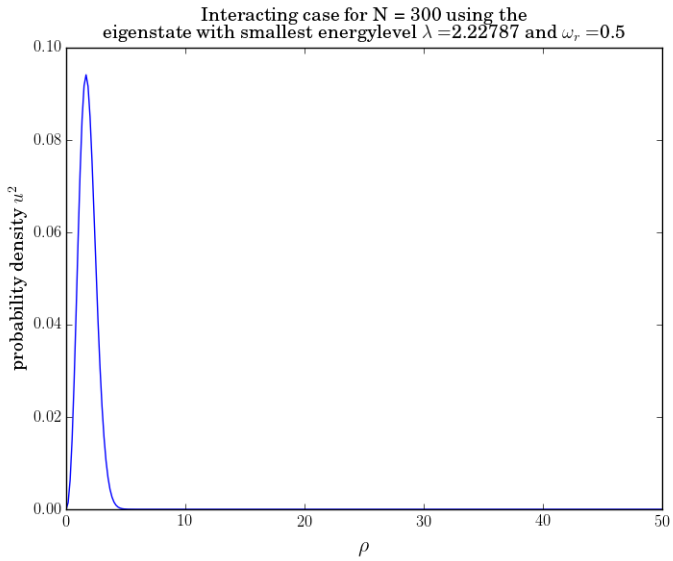
5 Interacting case

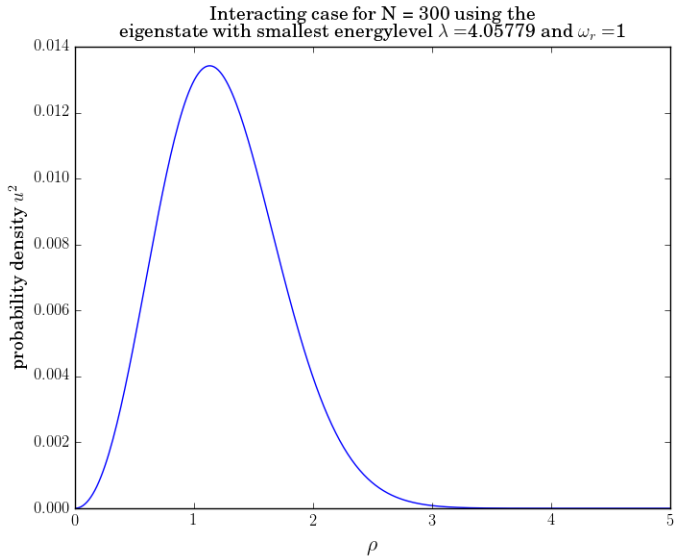
We have now seen that we can solve our system for the non-interacting case with a single electron. It is possible to use the same solver for the case when we have interaction between two electrons in an oscillator well. The system can be changed by simply changing the definition of V_i in 2. It can be show (see [3], page 5-6) that this is possible. We will now look at the cases with and without the Coulomb interaction:

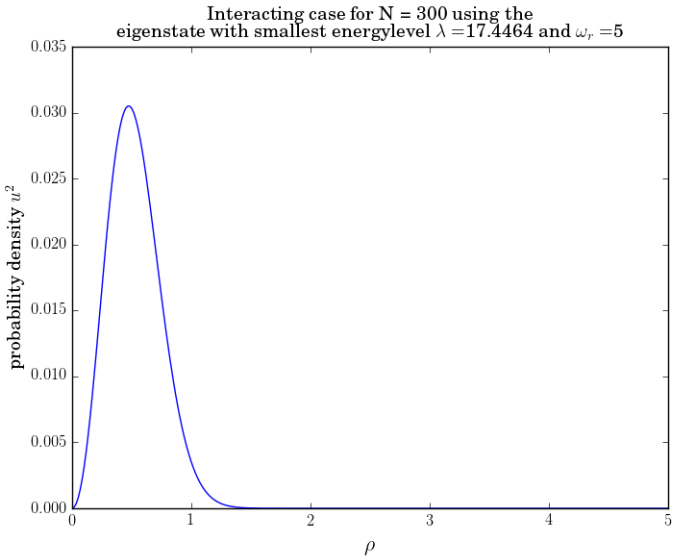
5.1 With Coulomb force

Here the potential becomes $V_i = \omega_r^2 \rho_i^2 + \frac{1}{\rho_i}$ with Coulomb force $\frac{1}{\rho}$ in 2. Note that we focus still on the ground state our electrons. We will solve the systems for $\omega_r = 0.01, \omega_r = 0.5, \omega_r = 1 and \omega_r = 5$. The values are chosen that we can see the behavior of the system of values between two extremes; with weak influence in the electrons from the oscillator well to strong influence.
We have the following plots after solving the system for each ω_r:
Here we will see that for high ω_r, $\omega_r = 1$, $\omega_r = 5$, the electrons are forced to be closer together. Since we are







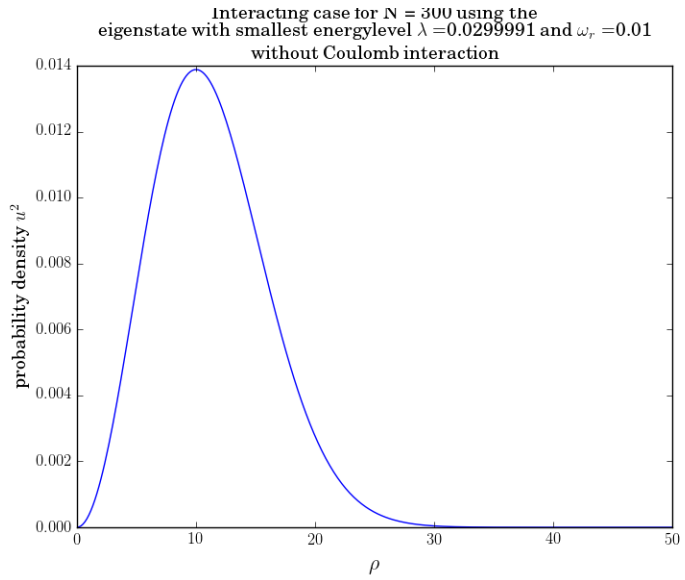


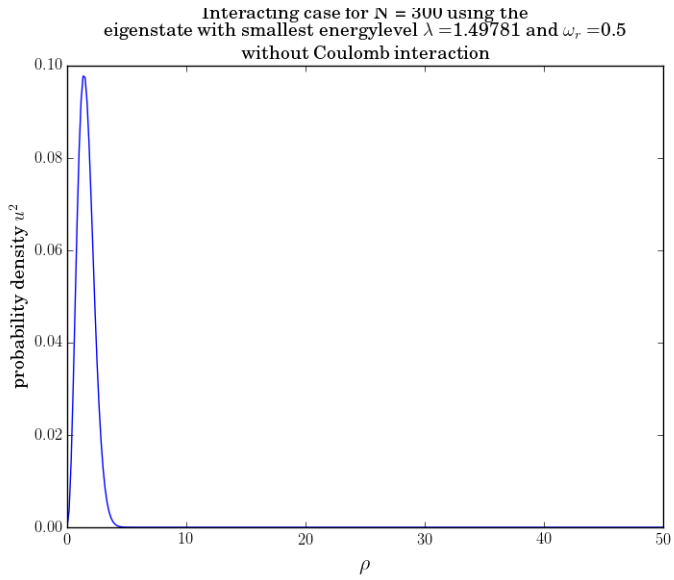
Plots fo the probaility density functions for the lowest energy levels for different values of ω_r. Note that in the upper rows, $\rho_{n-1} = 50$, and in the lowest $\rho_{n-1} = 5$. This is due to the electrons will be more 'spread' out in the well because of the weak interaction the well has with the electrons. There is therefore necessary to extend ρ such that we don't force the probability density to be zero.

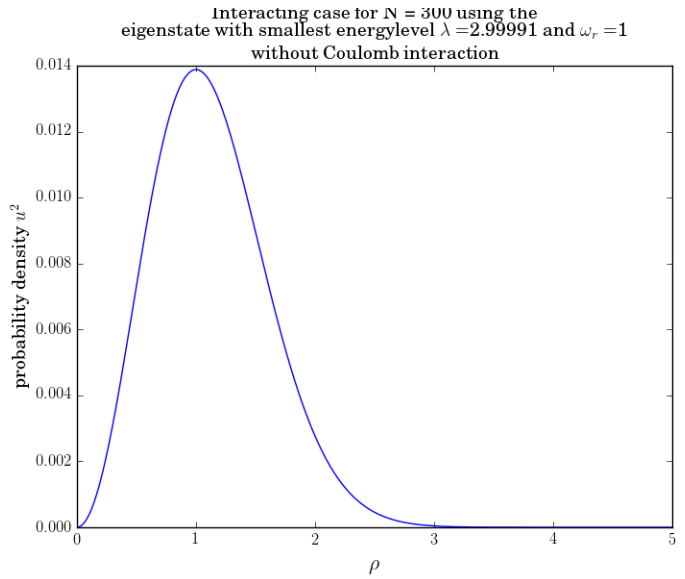
looking at the eigenstate with the lowest energy level, we would expect the electron to be around the bottom of the well when the oscillator frequencies are high. However, at low frequencies (especially$\omega_w = 0.01$) and due to the repulsive Coulomb force we see that the electrons are more spread and therefore give us a wide probability density. The high frequencies force the electrons to be closer, and therefore making the width of the probability density smaller.

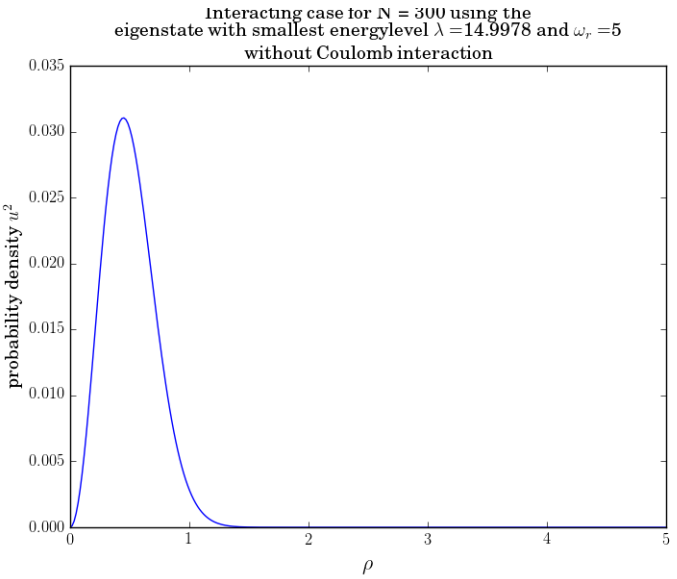
5.2 Without Coulomb force

Here it is enough to run our system with the potential $V_i = \omega_r^2 \rho^2$, that is the without the term describing the Coulomb force $\frac{1}{\rho}$. The following plots are the results for solving our system with the same values for ω_r as in the case with Coulomb force. Here we see a slightly different behaviour of our electrons with the repulsive









forces. We see that the propability density is much narrower than in the case with Coulomb interaction. This is expected, since we have not any repulsive forces between the electrons and will therefore tend closer to each other. This can especially be seen at the case $\omega_r = 0.01$

6 Conclusion

We have developed an eigenvalue solver to solve for the Schrödinger's equation for one and two interacting electrons. The solver is considerably slow compared to `numpy.linalg.eig`. This was expected since Jacobi's rotation method has low convergence rate. For the resulting solver, we have to be careful with our choice of mesh points n and for which value we must set ∞, ρ_{n-1} when we have to solve for the interactive and non-interacting case.
We have also shown how to discretize Schrödinger's equation and with some assumptions on our system, we can rewrite the eqution into an eigenvalue problem, both for the interactive and the non-interactive case.
Looking toward the future, the Jacobi's method is a method which should be avoided to use for large systems. Other methods, such as the QR-decomposition should be considered instead. However, the solver can give us accurate results depending on our choices of n and ρ_{n-1}.

References

[1] Quantum harmonic oscillator. `https://en.wikipedia.org/wiki/Quantum_harmonic_oscillator`.

[2] M. Hjorth-Jensen. *Computational physics, lecture notes*. 2015.

[3] M. Hjorth-Jensen. Project description. `https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Projects/2016/Project2/project2_2016.pdf`, 2016.

Appendix A: Tables for the non-interacting case

Here we try to find the values of n and ρ_{n-1} which gives us the best results. The aim is to get computed eigenvalues with four leading digits after the decimal point (by considering the values to be in scientific notation). The exact eigenvalues are $\lambda_0 = 3, \lambda_1 = 7, \lambda_2 = 11$.

Table 1: Three lowest computed eigenvalues for $\rho_{n-1} = 1$

n	λ_0	λ_1	λ_2
25	10.137070	39.574412	88.018665
75	10.149681	39.775680	89.034335
175	10.150896	39.795103	89.132627
300	10.151073	39.797940	89.146988
400	10.151113	39.798577	89.150212

Here we can see that there is no point in increasing n to get better results.
The value of ρ_{n-1} is therefore too low.

Table 2: Three lowest computed eigenvalues for $\rho_{n-1} = 3.5$

n	λ_0	λ_1	λ_2
25	2.994151	7.011897	11.407615
75	3.000101	7.042969	11.505857
175	3.000673	7.045947	11.515300
300	3.000756	7.046382	11.516678
400	3.000775	7.046480	11.516988

Here we can see that the eigenvalues are starting to be approximately the same as the exact.
However, we can see that the values for λ_2 differ too much from 11

Table 3: Three lowest computed eigenvalues for $\rho_{n-1} = 5$

n	λ_0	λ_1	λ_2
25	2.986369	6.931499	10.832098
75	2.998573	6.992862	10.982763
175	2.999742	6.998712	10.997048
300	2.999913	6.999565	10.999131
400	2.999951	6.999757	10.999599

Now the values seems to converge to the exact values.
The precision is approximately four leading digits after the decimal point at n = 400.
We have that n = 400 and $\rho_{n-1} = 5$ gives us results with good precision.

Table 4: Three lowest computed eigenvalues for $\rho_{n-1} = 6.5$

n	λ_0	λ_1	λ_2
25	2.976884	6.883389	10.712673
75	2.997587	6.987923	10.970505
175	2.999564	6.997819	10.994677
300	2.999852	6.999261	10.998198
400	2.999917	6.999585	10.998988

We can see that for n = 400 and n =300, the precision is not as good as for $\rho_{n-1} = 6.5$.
We can therefore conclude that n = 300 or n = 400 for $\rho_{n-1} = 5$ gives the best results.

Appendix B: Time comparison, Jacobi and numpy.linalg.eig

The program which sets up the system and then calls `numpy.linalg.eig` is taken from **here** (at the bottom of the page) which some modifications, see `eig.py`. Here are the results after calling `run_non_interacting` in `project2.py`. The system solved and taken time for is for the non-interacting case.

```
Time used for non-interacting with N = 75:
            - solver from jacobi.cpp used 0.1457 seconds
            - numpy.linalg.eig from eig.py used 0.1163 seconds

Time used for non-interacting with N = 175:
            - solver from jacobi.cpp used 4.1019 seconds
            - numpy.linalg.eig from eig.py used 0.1130 seconds

Time used for non-interacting with N = 300:
            - solver from jacobi.cpp used 35.1530 seconds
            - numpy.linalg.eig from eig.py used 0.3143 seconds

Time used for non-interacting with N = 400:
            - solver from jacobi.cpp used 116.7054 seconds
            - numpy.linalg.eig from eig.py used 0.6180 seconds

Time used for non-interacting with N = 650:
            - solver from jacobi.cpp used 875.4539 seconds
            - numpy.linalg.eig from eig.py used 1.8550 seconds
```

The file generated from `run_non_time` in `project2.py`. Here we can see that the Jacobi rotation method is much slower compared to `numpy.linalg.eig`, especially for high values on n.