Project 2

Jialin Liu
Thanaphong Phongpreecha

March 6, 2016

Abstract

In this work, we aim to solve one and two electron Shrodinger's equation by converting it to eigenvalue problem. The main algorithm used is Jacobi's method, along with other built-in functions in Python, i.e. *eig* and *eigh*. When compared to built-in function in Python, our in-house code performs much slower (4 orders of magnitude). The results showed that using ϵ of 1E-04 is enough and give almost the exact same result as 1E-14. The time and transformation required increase as n steps increase, with time increasing more drastically. The choice of ρ_{max} significantly affects eigenvalues where the smaller ρ_{max} gives higher energy (eigenvalues). In two electrons system, a frequency w_r corresponding to the strength of potential well is defined. With higher w_r and its impact on electron probability and shape of wave function are discussed. The effect of repulsion between two electrons are also demonstrated.

1 Introduction

1.1 General Background

After the introduction of Schrödinger equation one hundred years ago, only a few systems can be solved analytically such as isolate hydrogen atoms or electrons confined in infinite potential well due to its partial differential equation nature. To calculate the energy level and wave function for a more complicated system, numerical methods have to be introduced in order to solve the simplified time-independent non-relativistic Schrödinger equation.

In this work, we converted the Schrödinger equation of one and two electrons in harmonic oscillator potential to a eigenvalue problem and implemented the Jacobi Rotation method to numerically give the solution of eigenvalue and eigenfunctions under different parameters such as radius ,strength of potential and the existence of Coulomb interaction.

In following content of section 2, the mathematical assumptions and derivation are explained for solving one electron and two electrons Schrödinger equation. Also, the Jacobi Rotation Method are explained. In section 3, different modules in implementation of Jacobi Rotation Method in Python are described. To calibrate the algorithm, individual unit test are done and the results are shown here. The calculation results and its discussions are shown in Section 4.

1.2 One electron Schrödinger equation

With an interest in the single electron equation (with a potential term V),

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

Although detailed derivation can be found in the problem statement, the above equation was translated to a solvable eigenvalue problem by mean of dimensional analysis and, for this case, setting $l=0$ for ground state calculation. This ultimately resulted in

$$-\frac{d^2}{d\rho^2}u(\rho) + \frac{mk}{\hbar^2}\alpha^4\rho^2 u(\rho) = \frac{2m\alpha^2}{\hbar^2}Eu(\rho).$$

The constant α can now be fixed so that

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

or

$$\alpha = \left(\frac{\hbar^2}{mk}\right)^{1/4}.$$

Defining

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

we can rewrite Schrödinger's equation as

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

This is the first equation to solve numerically. In three dimensions the eigenvalues for $l = 0$ are $\lambda_0 = 3, \lambda_1 = 7, \lambda_2 = 11, \dots$.

The second derivative can be rewritten as,

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

The equation becomes,

$$-\frac{u(\rho_i + h) - 2u(\rho_i) + u(\rho_i - h)}{h^2} + \rho_i^2 u(\rho_i) = \lambda u(\rho_i),$$

where,

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

$$\rho_i = \rho_{\text{min}} + ih \qquad i = 0, 1, 2, \dots, n_{\text{step}}$$

and $V_i = \rho_i^2$ is the harmonic oscillator potential. Define first the diagonal matrix element

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

and the non-diagonal matrix element

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

1.3 Two electrons Schrödinger equation

When two electrons exist in the harmonic oscillator well, the Schrödinger equation will be rewrite to the following form:

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

Using similar substitution and dimensionless methods, the two electrons Schrödinger equation can be transform to the following form:

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

where

$$\rho = r/\alpha,$$

and

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

To further simplify the equation, we introduce the relative coordination $r = r_1 - r_2$ and center-of-mass coordination $R = 1/2 * (r_1 + r_2)$. Also, if we consider the repulsion between two electrons, an extra potential term $V(r_1, r_2)$ need to be added as follow form:

$$V(r_1, r_2) = \frac{\beta e^2}{|\mathbf{r}_1 - \mathbf{r}_2|} = \frac{\beta e^2}{r},$$

So, the Schr$\acute{}$odinger equation become:

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

Defining

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

and substitute in all the varibles above, the final form of Schr$\acute{}$odinger equation become:

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

1.4 Jacobi Rotation Method

By implementing the numerical Euler's Scheme, the above mentioned Schr$\acute{}$odinger Equation can be discretized to the following form:

$$Ax = Ex$$

where A is a tridiagonal matrix, x is a vector and E is a scalar. So the ODE is transformed into an eigenvalue problem. Jacobi Rotation Method is popular method that converts a symmetrical matrix to diagonal matrix. And the eigenvalues are simply the diagonal elements. In each iteration of Jacobi Rotation Method, the non-diagonal element with largest absolute value, $A(p, q)$ is selected and correspond rotation matrix S is generated such that the $S(p, p) = cos\theta$ and $S(q, q) = cos\theta$. The rest elements are shown below:

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & 0 & \dots & \dots & 0 \\ 0 & cos\theta & 0 & 0 & sin\theta & \dots \\ \dots & 0 & 1 & 0 & 0 & \dots \\ 0 & \dots & \dots & 1 & \dots & \dots \\ 0 & -sin\theta & 0 & 0 & cos\theta & 0 \\ 0 & \dots & 0 & 0 & 0 & 1 \end{pmatrix} \quad (1)$$

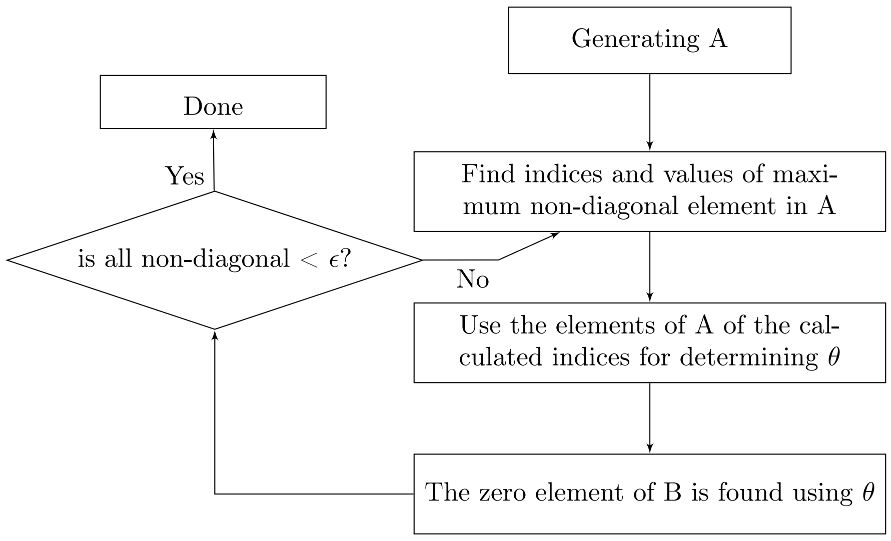
To rotate A matrix,we need to left multiple S^T and right multiple S. The value of θ can be found by assign 0 to the (p, q) element in the matrix after rotation. Correspondingly, the eigenvector need to left multiply S^T.

In each iteration, the non-diagonal element with maximum absolute value is picked to generate rotation matrix S and corresponding rotation angle. Eventually, the matrix will be transformed to a diagonal matrix.

2 Methods

2.1 Implementation of Jacobi's rotation algorithm in Python

As we are working in team, the algorithm was splitted into four main parts to distribute the workload. Then these parts were compiled together. The name of each section will be the main goal of each function. Generally, the function for Jacobi's algorithm will follow this flow chart:



Generating A
Done
Yes
Find indices and values of maximum non-diagonal element in A
is all non-diagonal < ϵ?
No
Use the elements of A of the calculated indices for determining θ
The zero element of B is found using θ

Figure 1: Algorithm chart for implication of Jacobi's method.

2.1.1 Generating Matrix A

The goal is to generate a matrix

$$A = \begin{pmatrix} \frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{n_{\mathrm{step}}-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{n_{\mathrm{step}}-1} \end{pmatrix} \tag{2}$$

To do this we rely on for loop function where a user specify size of the matrix (n), maximum dimensionless radius ρ_{max}, as inputs, for which we wrote as:

```
    for i in range(n):
        rho[i] = rho_min + i*h
        V[i] = np.square(rho[i])
        d[i] = 2/np.square(h) + V[i]

    A = np.identity(n)*d + np.diag(e,-1) + np.diag(e,1)

    return A
```

2.1.2 Finding Maximum Elements in A

The next step is to identify indices (p, q) within matrix A that contain the element of highest absolute value. Here in the code (p, q) are represented by (i, j). This function return the indices of the maximum element as well as its value to be used further. Also, since we do not to alter the diagonal elements, the maximum positions are found by copying matrix A, then zero out the diagonal before implementing algorithm to locate the maximum element.

```
    n = int(np.sqrt(A.size))
    A_copy = copy(A)
    np.fill_diagonal(A_copy,0)
    A_copy = np.abs(A_copy)
    Max = np.argmax(A_copy)
    Max_ij = np.unravel_index(Max, (n, n))
    i = Max_ij[0]
    j = Max_ij[1]
    Max = [i,j, A[i,j]]

    return Max
```

2.1.3 Solving for θ

The indices found in the previous part is used here to determine the theta for rotation. Since the values will be used in a form of sine and cosine in the next part, they are the outputs from this code.

Note that the formula is modified slightly from what stated in the previous section. This is to avoid possible error when large value of τ is obtained.

```
        if A[p,q] != 0:

         tau = (A[q, q] - A[p, p]) / (2 * A[p, q])

         if tau >= 0:
             t = 1.0 / (tau + np.sqrt(1.0 + tau * tau))
         else:
             t = -1.0 / (-tau + np.sqrt(1.0 + tau * tau))
         c = 1/np.sqrt(1+t*t)
         s = c*t
    # theta = np.arctan(t)
    else:
        s = 0.0
        c = 1.0
    theta = [s, c]
    return theta
```

2.1.4 Calculation of Matrix B

In each iteration, the elements of B are recalculated using the sine and cosine value of the θ determined earlier.

```
    B = copy(A)
    sizeA = np.sqrt(A.size)
    s = theta[0]
    c = theta[1]
    B[p, p] = A[p, p]*c*c-2*A[p, q]*c*s+A[q, q]*s*s
    B[q, q] = A[q, q]*c*c+2*A[p, q]*c*s+A[p, p]*s*s
    B[p, q] = 0
    B[q, p] = 0
    for i in range(0, int(sizeA)):
        if (i != p) and (i != q):
            B[i, p] = A[i, p]*c - A[i, q]*s
            B[p, i] = B[p, i]
            B[i, q] = A[i, q]*c + A[i, p]*s
            B[q, i] = B[i, q]

    return B
```

2.1.5 Calling All Functions in Loop to Obtain Diagonal Matrix

Although the elements in B is recalculated from θ, not all of them will be below the tolerance (ϵ) for equivalence of zero. However, it should be approaching and therefore all the previous codes are performed together in a *while* loop until all non-diagonal elements of B are virtually zero (i.e. $<$ err or ϵ).

```
start = time.clock()                # Start counting time
while np.abs(maxValue) > err:

    theta = solveTheta(A, maxPosition[0], maxPosition[1])
    B = CalculateB(A, theta, maxPosition[0], maxPosition[1])
    EigV = CalculateEigV(EigV, theta, maxPosition[0],
    maxPosition[1])
    A = B
    maxReturn = A_max(A)
    maxValue = maxReturn[2]
    maxPosition = maxReturn[0:2]
    count += 1

finish = time.clock()
EigenValues = sorted(np.diag(B))
```

```
print "Number of transformation = ", count
print "Three lowest eigenvalues = ", EigenValues[0:3]
print "The time used for calculation is ", finish-start
```

2.2 Unit test

The unit test are done under the frame of python unit test framework. Corresponding codes are placed under UnitTest folder in UnitTest.py file. All modules for Jacobi Rotation Method are tested individually.

For constructA module, a 3 by 3 matrix and 4 by 4 matrix are calculated by hand and hard coded in the test case. The absolute difference of norm between hand calculated and Python code generated is allowed to be smaller than 1E-6.

For maxA module, the test of ability to exclude diagonal elements and to compare the absolute value while return the true value are tested.

For solveTheta module, a random matrix element in a random 3 by 3 matrix is used. The result is compared with the hand calculated result.

For calculateB module, together with solveTheta module, after pick a random element in a 3 by 3 matrix, this element should become zero.

3 Results and Discussion

3.1 Explaining the Choice of t and Its Effects on Minimizing the Differences in Matrix B and A

As can be seen in the codes shown in the Method section, the equation for solving t is quadratic, and therefore, has two roots. The θ of the smaller root should be selected because given that

$$||\mathbf{B}-\mathbf{A}||_F^2 = 4(1-c)\sum_{i=1,i\neq k,l}^{n}(a_{ik}^2+a_{il}^2)+\frac{2a_{kl}^2}{c^2}.$$

, the differences between A and B can be minimized by having larger c (cosine). Cosine is larger in smaller θ, which translate to smaller value of t, in the convinced range of $\theta \leq \pi/4$

3.2 Investigating Characteristics of Results and Eigenvalues

3.2.1 Showcasing the code and its calculation efficiency (time and number of transformation

First, we will look at the converging behavior of the problem. Shown in Figure 2, is the converging three lowest eigenvalues using ρ_{max} of 5 and ϵ of E-14. Judging from the % difference, it can be observed that the difference become <5 % when n = 100. Apparently, to most applications ϵ of E-14 is overtly excessive. But we do this in order to observe if there would be any difference when applying lower ϵ. Then we changed the ϵ to E-04, and found that the results are virtually the same (results not shown). For example using both ρ_{max} and ϵ of 5, the E-14 yields , while E-4 yields . This represents an unnecessary accuracy, which comes with an expensive temporal cost. As shown in Figure 3, the difference in the time required between E-04 and E-14 criteria grows exponentially as n increases. To this yield, it should be noted that even with E-04, using n of more than currently showed results in incommensurate amount of time required for calculation.

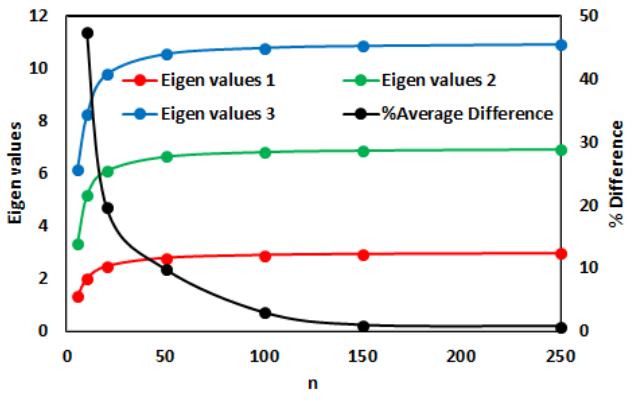


Figure 2: Converging behavior of the three lowest eigenvalues of $\rho_{max} = 5$ and ϵ = E-14 as a function of n. The % difference is calculated by comparing to the previous point.

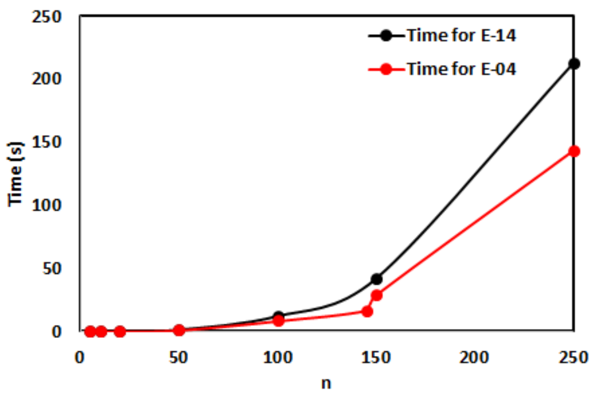


Figure 3: Calculation times for different ϵ.

Next, we will look at the number of transformation required in the Jacobi's algorithm as a function of n. Here we found that a function of time increment as a function of n steps rises much faster than those by number of transformation (Figure 4).

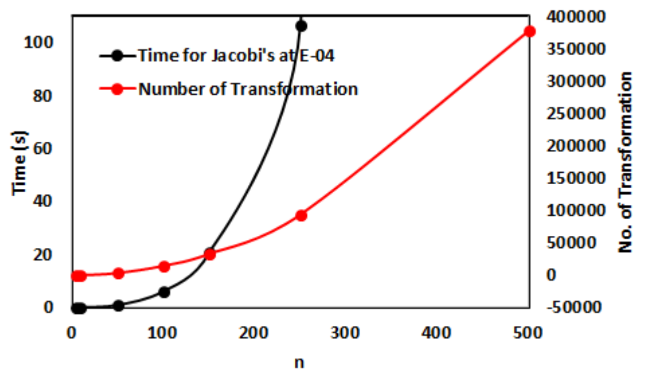


Figure 4: Calculation times for different ϵ.

3.2.2 Comparison with built-in algorithm in Python

In python, there are couple functions dedicated to solving eigen problems. These include $eigh, eig$. We performed two tests. Unfortunately, although with great effort, we were unable to find the algorithm used behind either of these built-in functions. However, we do know that eigh is dedicated for symmetric or Hermitian matrix, where as eig is an algorithm applicable to any general matrices. Both of these functions return eigenvalues and eigenvectors of virtually the same values as our code as well. We aim to investigate the time usage for each function as compared to our algorithm. from Figure 5 we can see that eigh which is built specifically for symmetric matrices have a superior speed than all other functions, whereas our in-house code perform poorly, and require time 4 order of magnitude more at n = 250. For this reason, many subsequent investigations in this report will utilize eigh function so that we can probe into higher n with less time.

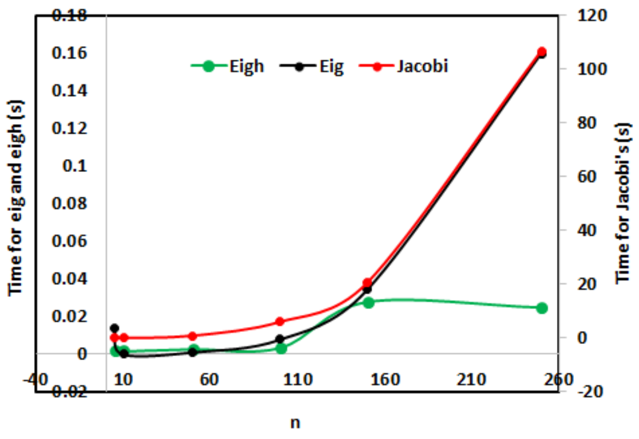


Figure 5: Calculation times for different coding method. Jacobi's represents the algorithm written by us

3.2.3 How many n steps are needed to get three eigenvalues of the same 4 leading digits? and dependence of eigenvalues on ρ_{max}

First of all, the values in the Table 1 is generated from a standard Python library because the we aim to have as many n steps as possible, so that the value is most accurate. To do so, the Jacobi's algorithm used in this class is not fast enough as demonstrated in the previous subsection.

Using the criteria of having the same 4 leading digit as a convergence, we found that it depends on the step size (i.e. h, which is dependent on both n and ρ_{max}). If the step size is small, then the difference from one step to the other would be small. As shown in Table 1, it is found that for a 200 step difference, the satisfying eigenvalue can be obtained as soon as at n = 2200. To be clear, it could have had reach the criteria of same 4 leading digits earlier if we took a smaller n difference, i.e. 200 then 205. This depends on the choice of the user.

As the step size h is also affected by ρ_{max}, the bigger ρ_{max} the larger step size it is for a certain number of n. The bigger step size implies that it will be more difficult reach the 4-leading digit criteria. Table 1 supported this argument. It can be seen that using ρ_{max} of 0.1 converged quickest. Although there is no big difference when increases ρ_{max} up to at least, a stark increment of ρ_{max} to 50 confirms the hypothesis as no value met the criteria for eigenvalue convergence.

Regardless of n, choosing ρ_{max} can have a tremendous impact on the resulting eigenvalues. Physically, if we were to confine an electron in a smaller radius (ρ_{max}), then one would imagine that a higher energy would be required to build a higher energy wall. This is illustrated in the Table 1, in which smaller ρ_{max} brings about larger eigenvalues. This behaves in an exponential manner. If we were to shrink ρ_{max} to 0.02, the converged (meeting 4 leading digit criteria) eigenvalues would become strikingly as high as 24657.56774, 98630.24363, 221917.9467.

Vice versa, if we allow electrons to move freely in a wider space, then as $\rho_{max} \to \infty$ the eigenvalues would $\to 0$. This makes sense as we do not need high energy to constrain electrons as the distance allowed become bigger.

3.2.4 Rounding Errors

Due to time limitation by our Jacobi's algorithm, the error tests were done by using the fastest algorithm *eigh*. Using this algorithm we found that the rounding error occurs at almost at the maximum memory of this computer. From Table 2, it can be seen that at n = 15000, the value as become exceedingly off that we cannot illustrated the data in a graph.

3.3 Results of two electrons system

Typical electron probability distribution with repulsion (system 1) are plotted in Figure 6. The wave function are converted to probability by multiply it self and normalize by its area.The unit for x axis is angstrom. As the figure shows, the ground state wave function have no nodes and have one maxima. The first and second excited state have two and three nodes, respectively. And they have more than one local maxima in the probability distribution.

Table 1: Variation of the three lowest eigenvalues as a function of n steps and ρ_{max}. The blue represents pairs of values that have the same 4 leading digits, while the red represent the first n step that all three eigenvalues reach the criteria.

n	**50**			**5**		
200	2.468	6.118	9.776	2.944	6.915	10.893
400	2.726	6.567	10.427	2.972	6.958	10.947
600	2.815	6.713	10.627	2.981	6.972	**10.965**
800	2.861	6.786	10.724	2.986	6.979	**10.974**
1000	2.888	6.829	10.781	2.989	6.983	**10.979**
1200	2.907	6.858	10.819	2.991	6.986	**10.983**
1400	2.920	6.878	10.845	2.992	6.988	**10.985**
1600	2.930	6.893	10.865	2.993	6.989	**10.987**
1800	2.938	6.905	10.880	2.994	6.991	**10.988**
2000	2.944	6.915	10.892	2.994	6.992	**10.990**
2200	2.948955	6.922683	10.9023	**2.994873**	6.992305	**10.99056**
2400	2.95319	6.929154	10.91056	**2.9953**	**6.992947**	**10.99137**
2600	2.956776	6.934626	**10.91753**	**2.995662**	**6.99349**	**10.99205**
2800	2.959851	6.939314	**10.92349**	**2.995972**	**6.993955**	**10.99263**
3000	2.962519	6.943374	10.92865	**2.99624**	**6.994358**	**10.99314**

n	**1**			**0.1**		
200	10.05083	39.403	88.255	977.1473	3908.342	8792.871
400	10.10086	39.601	88.707	982.0419	3928.099	8837.993
600	10.1176	39.667	88.857	983.6794	3934.682	8852.931
800	10.12598	39.700	88.931	984.4992	3937.974	8860.380
1000	**10.13101**	39.720	**88.976**	**984.9915**	3939.948	8864.843
1200	**10.13437**	39.733	**89.006**	**985.3198**	3941.264	8867.816
1400	10.13676	39.743	89.027	**985.5544**	3942.205	8869.938
1600	**10.13856**	39.750	89.043	**985.7304**	**3942.910**	8871.529
1800	**10.13996**	**39.755**	89.055	**985.8673**	**3943.458**	8872.766
2000	10.14108	**39.760**	**89.065**	**985.9768**	**3943.897**	8873.755
2200	10.142	**39.76334**	**89.07335**	**986.0665**	**3944.2558**	**8874.564**
2400	**10.14276**	**39.76634**	**89.08011**	**986.1412**	**3944.5549**	**8875.2383**
2600	**10.14341**	**39.76889**	**89.08582**	**986.2044**	**3944.808**	**8875.8087**
2800	**10.14396**	**39.77106**	**89.09072**	**986.2586**	**3945.0249**	**8876.2975**
3000	**10.14444**	**39.77295**	**89.09497**	**986.3055**	**3945.2129**	**8876.7211**

Table 2: Eigenvalues of different n for $\rho_{max} = 1$

n	**Rho = 1**		
5	6.904	25.260	50.267
50	9.756	38.211	85.456
500	10.111	39.641	88.797
5000	10.147	39.784	89.119
7500	10.148	39.789	89.131
10000	10.149	39.791	89.137
15000	-225000000.000	-225000000.000	-225000000.000

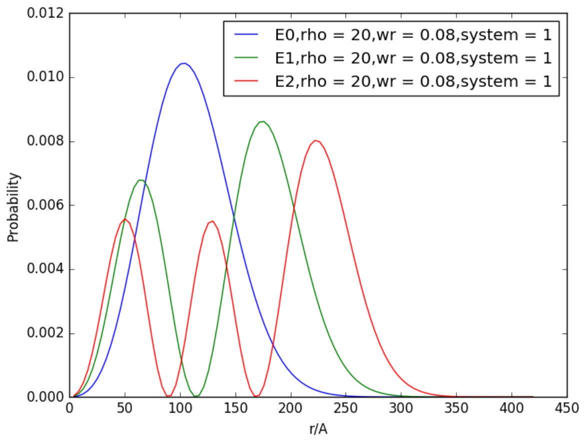


Figure 6: Probability distribution corresponding to three lowest eigenvalues for $\rho = 20$, $w_r = 0.08$ and with coulomb interaction

3.3.1 The impact of w_r and ρ on wave function

The ground state probability distribution of different w_r and ρ are calculated and shown in Figure 7 and Figure 8. w_r represents the strength of this potential well. So a larger w_r will confine the electron in a smaller region. Also, from the impact of repulsion shown in Figure 9 and 10, we can see that the peak shift to different position under different w_r value. This indicates the w_r parameter also impact the shape of wave function. Similarly, the ρ defines the boundary value of this wave function, so large ρ can allow wave function spread out. These results are consist with Taut[1]. The probability distribution of excited states are also calculated and they show same trend with ground state.

3.3.2 The impact of repulsion

The probability distribution of with repulsion (system 1) and without repulsion (system 2) are plotted in Figure 9 and Figure 10. A slightly change in the peak position can be observed. When two electrons have no repulsion with each other, the position of maximum probability will move to the middle.

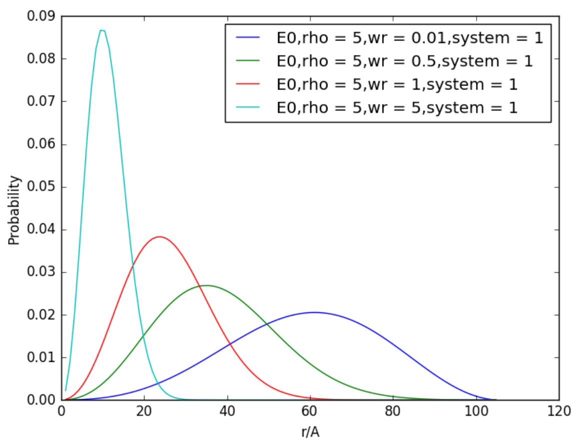


Figure 7: Probability distribution of ground state wave function with various w_r value with coulomb interaction

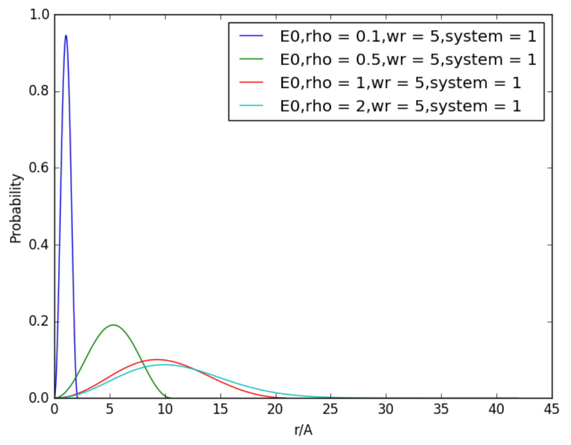


Figure 8: Probability distribution of ground state wave function with various ρ value with coulomb interaction

4 Conclusion

The Schrödinger equation for one and two electrons in harmonic oscillator potential well is derived and transformed to an eigenvalue problem of tridiagonal matrix using Euler Scheme. By implementing Jacobi Rotation Method, the tridiagonal matrix is transformed to a diagonal matrix. The convergence criteria and convergence speed are tested. $\epsilon = 1E - 4$ is enough for eigenvalues to be converge. The Jacobi Rotation Method is about 4 orders of magnitude slower than numpy eigenvalue solver *eig* and *eigh*. With decreasing $\rho_m ax$ and increasing w_r, the eigenvalue will increase and the position of maximum electron probability distribution will shift to origin. Also, the w_r will affect the shape of

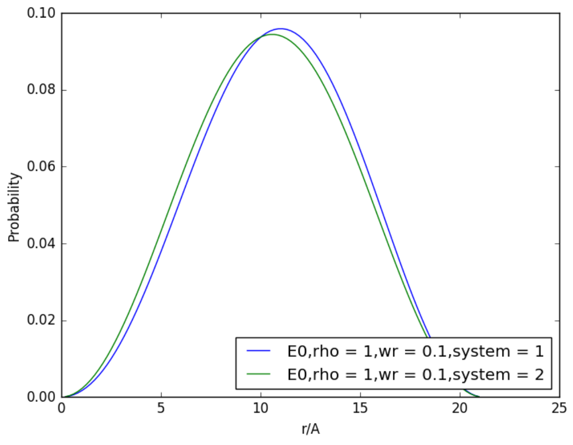


Figure 9: Probability distribution of ground state wave function with and without coulomb interaction with $w_r = 0.1$

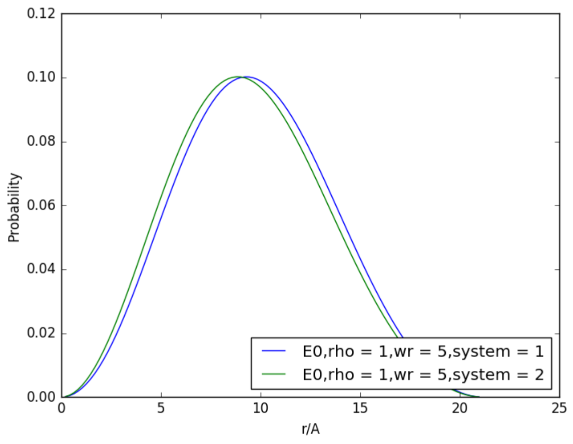


Figure 10: Probability distribution of ground state wave function with and without coulomb interaction with $w_r = 5$

wave function.

5 Reference

1. Taut, M. "Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem." Physical Review A 48.5 (1993): 3561.