Solution to Radial Schrödinger in a 3D Harmonic Oscillator
Computational Physics-Phy905
Project 2

Crispin Contreras

March 8, 2016

Abstract

This paper discusses the numerical solution for the radial Schrödinger equation in a 3D harmonic potential for two electrons. I consider the interaction and non-interaction of the electrons. I solved the equation with two different methods, the Jacobi method and the Armadillo eigenvalue solver. I compared the computational time for both methods and found that the Armadillo method is much faster than the Jacobi method. For the non-interacting case we know the analytical solution and for the interacting case I compared to the values used in [2].

1 Introduction

We begin with the radial Schrödinger equation with a 3D harmonic oscillator potential. After some manipulation we get the equation to have less physical constants and also simplify it. The second derivative of the equation is approximated by the three point formula and we obtain a set of linear equations. We put them in matrix form which gives us the equation $\mathbf{Av} = \lambda\mathbf{v}$. By writing it in this form we find that $\mathbf{A}$ is a tridiagonal matrix. This whole process is discussed in the Theory section. The Jacobi and Armadillo's eigenvalue solver are then used to solve for the eigenfunction $\mathbf{v}$ and eigenvalues. A discussion of how the Jacobi Algorithm works in given in the Methods section. Only the three lowest eigenfunctions and eigenvalues which are $\lambda = 3, 7, 11$ were solved for. The precision of these values changed as the number of grid points (n) increased. Finally I discuss my results in the discussion section.

2 Theory

2.1 Single Electron

We start with the radial part of the Schrödinger equation for one electron in a 3D harmonic oscillator. The general equation is

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

In our case the potential $V(r) = \frac{1}{2}m\omega^2 r^2$ and the energy is given by $E = \hbar\omega\left(2n + l + \frac{2}{3}\right)$ where $n = 0, 1, 2, ...$ and $l = 0, 1, 2, ...$. Here l is the orbital angular momentum of electron

and n is the principal quantum number. The Schrödinger equation can be simplified by making the following substitution $R(r) = \frac{u(r)}{r}$ and we obtain

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

with the following boundary conditions $u(0) = 0 \quad and \quad u(\infty) = 0$. In order to make the algorithm smoother I introduce a dimensionless variable $\rho = \frac{r}{\alpha}$ and obtain the modified Schrödinger equation

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2u(\rho)}{dr^2} + \left(\frac{\hbar^2}{2m\alpha^2}\frac{l(l+1)}{\rho^2} + V(\rho)\right)u(\rho) = Eu(\rho)$$

I only look at the solutions where $= 0$ and plug in for the potential $V(\rho) = \frac{1}{2}k\alpha^2\rho^2$ and reach the form

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2u(\rho)}{dr^2} + \frac{1}{2}k\alpha^2\rho^2u(\rho) = Eu(\rho)$$

We now multiply both side by $\frac{2m\alpha^2}{\hbar^2}$ and obtain

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

The α constant can now be fixed so that

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

or

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

I also define

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

I arrive at the final form for the equation

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

This equation is now easier to handle since it has less physical constants. In three dimensions the eigenvalues for $l = 0$ are $\lambda_0 = 3, \lambda_1 = 7, \lambda_2 = 11, ...$

I then use the three point formula for the second derivative

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

where h is our step. This is defined as

$$h = \frac{\rho_m ax - \rho_m in}{n+1}$$

and the arbitrary value of ρ is

$$\rho_i = \rho_m in + ih \quad i = 0, 1, 2, .., n+1$$

the simplified Schrödinger equation now reads

$$-\left(\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}\right)+\rho_i^2 u_i=-\left(\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}\right)+V_i u_i=\lambda u_i$$

This equation can now be put in matrix form. The diagonal terms are given by $d_i=\frac{1}{h^2}+V_i$ and the non-diagonal terms are $e_i=-\frac{1}{h^2}$.

$$\begin{pmatrix} d_1 & e_1 & 0 & \dots & \dots & \dots & 0 \\ e_1 & d_2 & e_1 & 0 & \dots & \dots & 0 \\ 0 & e_1 & d_3 & e_1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & e_1 & d_{n-1} & e_1 \\ 0 & \dots & \dots & \dots & 0 & e_1 & d_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{pmatrix} = \lambda \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{pmatrix} \tag{3}$$

2.2 Two Electrons with interaction

We now consider two electrons in a harmonic oscillator potential. We start with the Schrödinger equation of the two electrons without interaction

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

where $u(r_1,r_2)$ is the wavefunction for the two electrons and $E^(2)$ is the respective energy. We know introduce a new set of coordinates which are $\mathbf{r}=\mathbf{r_1}-\mathbf{r_2}$ and the center of mass coordinate $\mathbf{R}=\frac{1}{2}(r_1+r_2)$. With some algebra I get that $\mathbf{r_1}=\frac{\mathbf{r}}{\mathbf{2}}+\mathbf{R}$ and $\mathbf{r_2}=\mathbf{R}-\frac{\mathbf{r}}{\mathbf{2}}$. Also the derivative

$$\frac{d}{dr_1}=\frac{d}{dr}\frac{dr}{dr_1}+\frac{d}{dR}\frac{dR}{dr_1}=\frac{d}{dr}+\frac{1}{2}\frac{d}{dR}$$

and taking the derivative again we obtain the final form

$$\frac{d}{dr_1}\left(\frac{d}{dr_1}\right)=\frac{d^2}{dr^2}+\frac{1}{4}\frac{d^2}{dR^2}+\frac{1}{2}\frac{d}{dR}\frac{d}{dr}+\frac{1}{2}\frac{d}{dr}\frac{d}{dR}$$

The same thing is obtain for r_2 except that the cross terms are negative. Plugging this into the 2 electron Schrödinger equation we obtain

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

Now we can separate the equations into $u(r,R)=\psi(r)\phi(R)$ doing this will give the energy as $E^{(2)}=E_r+E_R$ where E_r is the relative energy and E_R is the center of mass energy. We now introduce the Coulomb interaction between the electrons which is given by $V(r_1,r_2)=\frac{\beta e^2}{|\mathbf{r_1}-\mathbf{r_2}|}=\frac{\beta e^2}{r}$ where $\beta e^2=1.44eVnm$. Since this has the relative term r we can plug this into the equation. This means that we can ignore the center of mass equation for now. Plugging the Coulomb interaction I obtain

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

we introduce the dimensionless variable $\rho = \frac{r}{\alpha}$ to get to the same form as equation (1). We then get

$$\left(-\frac{d^2}{dr^2} + \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4\rho^2 + \frac{m\alpha\beta e^2}{\rho\hbar^2}\right)\psi(\rho) = \frac{m\alpha^2}{\hbar^2}E_r\psi(\rho) \tag{4}$$

we define a "frequency"

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

and fix the α by requiring that

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

or

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

and

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

Finally arrive at the final form

$$\left(-\frac{d^2}{d\rho^2} + \omega_r^2\rho^2 + \frac{1}{\rho}\right)\psi(\rho) = \lambda\psi(\rho) \tag{5}$$

We now treat ω_r as the strength of the potential and again consider only $l = 0$.

3 Methods

Two methods were implemented two solve the Schrödinger equation. One was the Jacobi method which consists of many similarity transformations and the other one was using Armadillo's eigenvalue solver which is called by $eig - sys$. I will discuss the Jacobi method here. I take much of the derivations from [1].

3.1 Jacobi Method

The Jacobi method consists of using many similarity transformation to reduce a matrix into diagonal form. In this case the matrix is $\mathbf{A}$. This method is chosen since doing a determinant for large matrices is impractical. The similarity transformation is defined by

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

The $\mathbf{S}$ has the following property $\mathbf{S^T} = \mathbf{S^{-1}}$. In our case the similarity transformation is defined by

$$s_{kk} = s_{ll} = cos(\theta), \quad s_{kl} = -s_{lk} = -sin(\theta), \quad s_{ii} = 1, \quad i \neq l \quad i \neq k$$

the other terms are zero. The results for the $\mathbf{B}$ matrix are

$$\begin{aligned}
b_{ik} &= a_{ik}cos(\theta) - a_{il}sin(\theta), \quad i \neq k, \quad i \neq l \\
b_{il} &= a_{il}cos(\theta) + a_{ik}sin(\theta), \quad i \neq k, \quad i \neq l \\
b_{kk} &= a_{kk}cos^2(\theta) - 2a_{kl}cos(\theta)sin(\theta) + a_{ll}sin^2(\theta), \\
b_{ll} &= a_{ll}cos^2(\theta) + 2a_{kl}cos(\theta)sin(\theta) + a_{kk}sin^2(\theta), \\
b_{kl} &= (a_{kk} - a_{ll})cos(\theta)sin(\theta) + a_{kl}(cos^2(\theta) - sin^2(\theta))
\end{aligned} \tag{6}$$

The recipe is the to chose θ so that all non-diagonal elements b_{kl} become zero. We require that $b_{kl} = b_{lk} = 0$ which then leads to

$$b_{kl} = (a_{kk} - a_{ll})cos(\theta)sin(\theta) + a_{kl}(cos^2(\theta) - sin^2(\theta)) = 0$$

if $a_{kl} = 0$ then this leads to $cos(\theta) = 1$ and $sin(\theta) = 0$. To solve the equation above we define $tan(\theta) = t = \frac{s}{c}$, $sin(\theta) = s$, $cos(\theta) = c$, and use the trigonometric identity $cos(2\theta) = \frac{1}{2(cot(\theta) - tan(\theta))}$ to obtain

$$cos(2\theta) = \tau = \frac{a_{ll} - a_{kk}}{2a_{kl}}$$

using the trigonometric identity we obtain

$$t^2 + 2\tau t - 1 = 0$$

resulting in

$$t = -\tau \pm \sqrt{1 + \tau^2}$$

and we can obtain c and s by using trigonometric tricks

$$c = \frac{1}{\sqrt{1 + t^2}}$$

and $s = tc$. In order to get the off-diagonal terms to be equal to zero or in our case $\geq 10^{-8}$ we have to complete many similarity transformations until this occurs. In order to figure this out we look at the Frobenius norm which is defined as

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}|a_{ij}|^2}$$

One of the important properties of the Frobenius norm is that it's the same after similarity transformations. This property will be used in the Jacobi algorithm. Solving the quadratic equation we find that $|\theta| \leq \frac{\pi}{4}$ having this property then leads to a minimization of the difference between $\mathbf{A}$ and $\mathbf{B}$ since

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

To summarize, the Jacobi methods consists of many similarity transformations in order to decrease the value of the off diagonal elements. This can be done by looking at the norm. This algorithm was implemented in C++ and it's show below in listing 1.

```
1  //Implementing Jacobi Method
   double  max = fabs(A(0,1));
3  int l; //indices
   int k;
5
   clock_t start, finish;
7  start = clock();

9  while(max > epsilon)
```

```
{
  max =0.0;
        for (int i = 0 ; i< (n-2) ; i++)
        {
                    for(int j =i+1 ; j<(n-1); j++)
                    {
                        if (fabs(A(i, j)) > max)
                        {
     max =fabs(A(i,j));

    //Find the indices
                            k=i;
                            l =j;
                        }
                     }
                }//End of i loop

  //Call function to rotate
  rotate(A,R,k,l,n);

  iterations++;
}
```

Listing 1: This shows how the Jacobi method was implemented. The rotate function is called here and it's responsible for carrying out the operation in (6).

4 Results

4.1 Single Electron

The Jacobi algorithm was implemented in C++ according to section 3. This was then compared to the Armadillo's eigenvalue solver $eig - sys()$ to make sure it was working properly. The energy eigenvalues are know and were discussed in the theory sections. Their values 3,7,and 11 were used for calibration for getting the ρ_{max}. By trial and error we found that the best value for the three lowest states was $\rho_{max} = 5$. The results for the eigenvalues and computational times are shown in table 1 below. From table 1 we can see that as the value of n increases the eigenvalues get closer to the correct values. Table 1 also shows the number of iterations that the Jacobi method must make in order for the off-diagonal terms to be $\leq 10^{-8}$. From table 1 we can also estimate the number of similarity transformations in order to get the off-diagonal terms to 10^{-8} which is roughly $1.7n^2$. Finally we made a plot using matplolib for the corresponding three lowest state which is shown in figure 1. In this case we made sure the wavefunctions were normalized as $\int_0^\infty |u(\rho)|^2 d\rho$. This was done by approximating the integral as a set of rectangles (like a histogram) and using the values of the normalized eigenfunctions as the height and the width from h we can calculate the area.The the integral is just

$$\int_0^\infty |u(\rho)|^2 d\rho \approx h\left(\sum_{j=1}^{n} |u_j|^2\right)$$

From this then we can get the normalized eigenfunctions by multiplying by $\frac{1}{h}$. This same procedure is done for the two electron case.

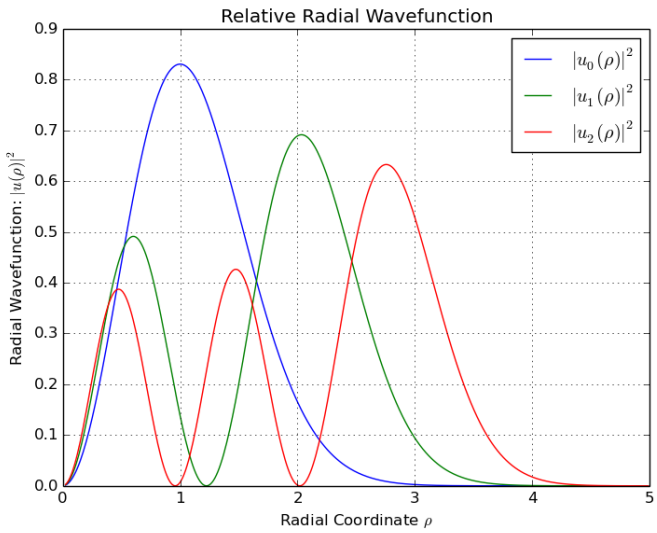

Figure 1: Plot of the radial wavefunctions for the three lowest energies. The eigefunctions were normalized and we used $\rho_{max} = 5$ and n=400.

Grid Points, n	3 Lowest Energy Eigenvalues For Jacobi, λ	Calculation Time, T_{Ja} [s]	Calculation Time, T_{arma} [s]	Number of iterations
50	2.9969, 6.9850, 10.9634	0.01653	0.001274	4040
100	2.9992, 6.9962, 10.9908	0.22181	0.003996	16475
200	2.9998, 6.9990, 10,9978	3.67701	0.021410	66828
300	2.9999, 6.9996, 10.9991	17.7895	0.069452	150798
350	2.9999, 6.9997, 10.9994	34.5932	0.083118	205757
400	3.0000, 6.9998, 10.9996	57.7010	0.122340	269021

Table 1: This table shows the number of grid points, the eigenvalues, Computational time, and the number of iterations. From here we can interpolate the number of similarity transformations needed to reach 10^{-8}.

4.2 Two Interacting Electrons

We use the same code for the single electron and modify it to include the Coulomb interaction. To see if the values are correct we use [2] and the values given by this source. This is shown in table 2.

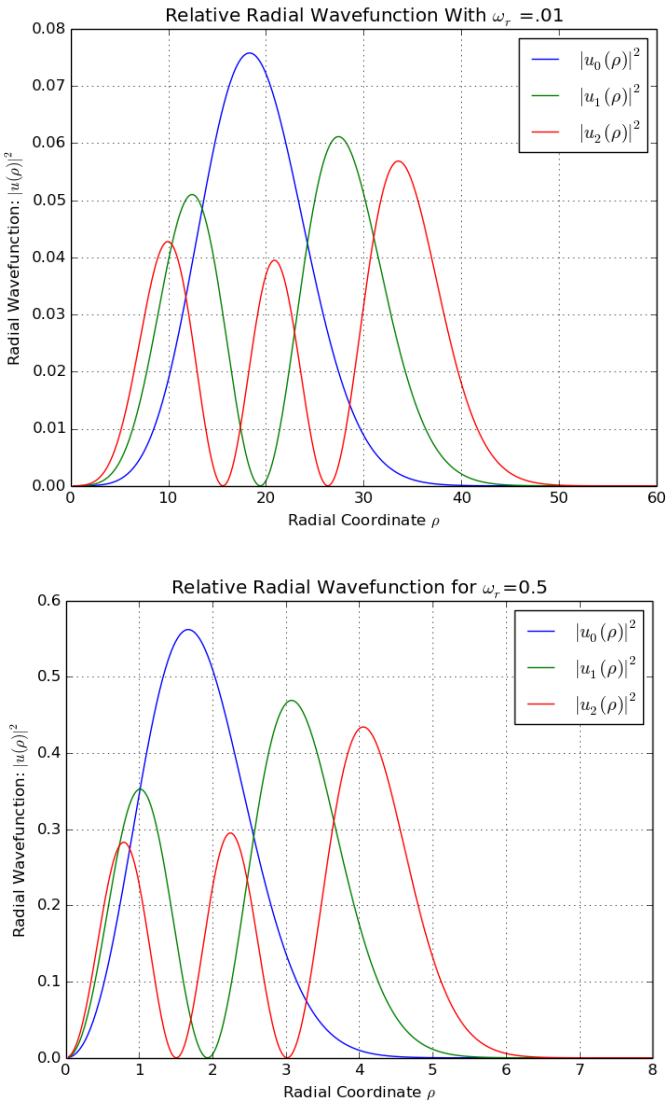

Figure 2: This figure show the normalized eigenfunction for ω_r =0.01 and ω_r=0.5. The values for ρ_{max} are 60 and 8 repectively witn $n = 400$.

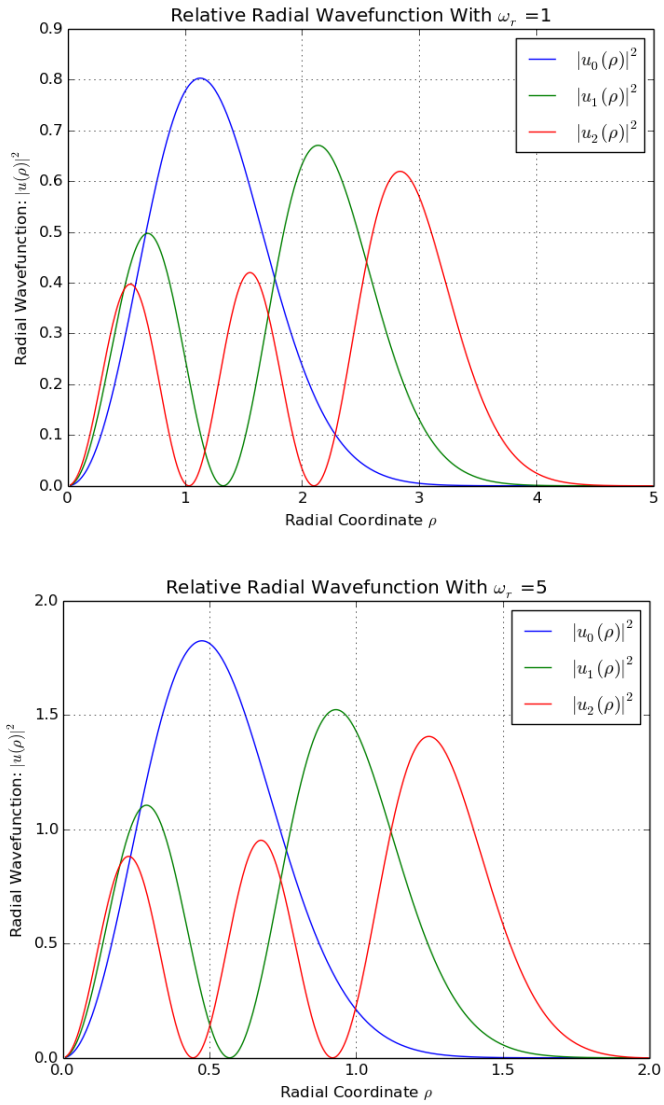

Figure 3: This figure show the normalized eigenfunction for ω_r =1 and ω_r=5. The values for ρ_{max} are 8 and 2 repectively witn $n = 400$.

Oscillator Frequency ω_r	Ground State Energy Eigenvalues, λ	[2] Approximate Formula, 2 ϵ'	[2]improved formula, 2 $\epsilon'_i nt$
0.01	0.1058	0.1050	–
0.05	0.3500	0.3431	0.3500
0.25	1.2500	1.1830	1.2500
0.5	2.2300	2.0566	–
1	4.0578	3.6219	–
5	17.4485	14.1863	–

Table 2: Values for the frequency ω_r and the energy eigenvalues. Also shown are the results from [2] which are the accepted values for the energies. We used these to make sure the algorithm was working correctly.

5 Discussion

The implementation of the Jacobi algorithm produced good results as shown in table 1 since they have good agreement with the know energy values. From table 1 we obtain that approximately $1.7n^2$ similarity transformations are needed to get the off-diagonal terms close to zero. This is less since our matrix is tridiagonal, for a full matrix a total of $3n^2$ to $5n^2$ transformations are needed[1]. We also saw that the Jacobi method has a very large computational time as opposed to the Armadillo's eigenvalue solver.

In trying to get reasonable results the biggest problem with this algorithm was determining the value of n and ρ_{max}. This took some time since it was done by trial and error. Although when I figured out one of the values it was easier to determine the others by changing ρ_{max} a bit. In order to start analyzing the interacting case we needed to use [2] to make sure the code was working correctly. For this I used the values for ω_r=0.25 and 0.05 and compared to [2]. As shown in table 2 our results show that our program was working correctly. In addition to this I also checked that the norm of the lowest eigenvectors were equal to 1. This is because the Frobenius norm is no affected by transformations. Again my results shows that this was true. These were the "unit test" that I implemented to make sure my code was working correctly.

Finally I discuss the physics of plots. I use figure 1 as basis and compared to the other plots. The main result is that the eigenfunctions were stretch or squeezed depending on the size of ω_r. For small values of ω_r we can see that eigenfunctions get stretched in the horizontal direction while for large values they get squeezed. We see that the electrons are further apart from each other when we include the interaction which is what is expected.

6 Conclusion

The Jacobi algorithm was implemented and we found that it had a very large computational time compared to Armadillo's eigenvalue solver. Additionally it was a bit time consuming figuring out what the best value for ρ_{max} was and similarly for n. The only way to do this was by trial and error. For the interacting case we found that the electrons get more spread out and this is what is expected since they have the same charges. The results for the interacting case were in agreement with [2] since the eigenvalues agreed.

7 References

[1] M. Hjorth-Jensen. Computational Physics, Lecture Notes Spring 2016.
[2] M. Taut.Two electrons in an external oscillator potential: Particular analytical solutions of a coulomb correlation problem. Phys. Rev. A. 48, 3561-3566 (1993)

8 Code Attachment

```
/*
  Author: Crispin Contreras
  Class: Physics 905
  Purpose: Solves the 3D radial Schrodinger equation in a harmonic potetial
  with two electrons. One way is using the Jacobi method taking into account
  the interaction between electrons and the other using the Armadillo library.

*/

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <stdio.h>
#include "armadillo"

using namespace arma;
using namespace std;
ofstream ofile;


void rotate(mat& , mat& , int , int , int);

int main()
{
  int n; //number of steps
  int iterations=0;
  int inter;
  char outfile[30]; //Rho and eigenfunctions
  char Eigen[30]; // Eigenvalues, elapsed time, iterations
  double epsilon = 1.0e-8;
  double rho_max, rho_min, w_r;
  double *rho_i, *V_Pot, *V_Pin;
  double h;

  //Read Input
  cout<<"Enter the number of steps n: ";
  cin>>n;
  cout<<"Enter the value of rho_max: ";
  cin>>rho_max;
  cout<<"Enter the value of rho_min: ";
  cin>>rho_min;
  cout<<"Do you want non-interacting(0) or interacting (1): ";
  cin>>inter;
  cout<<"Enter the value of the frequency w_r: ";
  cin>>w_r;
  cout<<"Enter the name of the outputfile for Rho and Eigenfunctions: ";
```

```
    cin>>outfile;
49  cout<<"Enter the name of file for Eigenvalues, elapse time, and total number
      of iterations: ";
    cin>>Eigen;
51


53  //calculate the step size
    h = (rho_max - rho_min)/n;
55
    //Allocate Memory
57  rho_i = new double [n+1];
    V_Pot = new double [n-1];
59  V_Pin = new double [n-1];

61  //Fillout value of potential
    for(int i=0; i<=n; i++)
63  {
       rho_i[i] = rho_min + i*h;
65
    }
67
    for (int i=0; i< (n-1); i++)
69  {
        //Non-Interacting Potential
71      V_Pot[i]=w_r*w_r*rho_i[i+1]*rho_i[i+1];

73      //Interacting Potential
        V_Pin[i] = V_Pot[i] + (1/rho_i[i+1]);
75
    }
77


79  //Declare variables
    mat A(n-1,n-1);
81  mat R(n-1,n-1);

83  //Filling the matrices
    A.zeros(n-1, n-1);
85  R.eye(n-1,n-1);

87  for (int i=0 ; i<(n-1); i++)
    {
89
        if(inter == 0)
91      {
           A(i, i) = 2/(h*h) + V_Pot[i];
93      }
        if(inter == 1)
95      {
  A(i,i) = 2/(h*h) + V_Pin[i];
97
        }
99
        if( i<(n-2))
101     {
  A(i,i+1) = A(i+1, i)= -1/(h*h);
103
```

```
        }
105
    }
107

109 //Copy for armadillo
    mat C = A;
111

113 //Implementing Jacobi Method
    double  max = fabs(A(0,1));
115 int l; //indices
    int k;
117
    clock_t start, finish;
119 start = clock();

121 while(max > epsilon)
    {
123   max =0.0;
          for (int i = 0 ; i< (n-2) ; i++)
125       {
                  for(int j =i+1 ; j<(n-1); j++)
127               {
                     if (fabs(A(i, j)) > max)
129                  {
       max =fabs(A(i,j));
131
      //Find the indices
133                      k=i;
                         l =j;
135                  }
                  }
              }//End of i loop

139   //Call function to rotate
      rotate(A,R,k,l,n);
141
      iterations++;
143 }

145 finish=clock();
    double time_j = (double (finish-start)/CLOCKS_PER_SEC);
147
    //Assorting the eigevalues and eigenvectors
149 vec Max(n-1);
    int Loc[3]={0,1,2};
151 double temp;

153 for(int i=0; i<(n-1);i++)
    {
155    Max(i)=A(i,i);
    }
157
    for(int i=0; i<(n-1);i++)
159 {
       for(int j=i+1; j<(n-1);j++)
```

```
161         {
                if(Max(i)>Max(j))
163             {
            temp=Max(i);
165         Max(i)=Max(j);
            Max(j)=temp;
167         if(i<3)
            {
169           Loc[i]=j;
            }
171             }

173         }

175     }

177     //Solving using Armadillo
              vec eigval(n-1);
179           mat eigvec(n-1,n-1);

181           start = clock();
              eig_sym(eigval, eigvec, C);
183           finish = clock();

185           double time_ar =(double(finish-start)/CLOCKS_PER_SEC);

187     //three lowest states Armadillo
        vec V0 = eigvec.col(0);
189     vec V1 = eigvec.col(1);
        vec V2 = eigvec.col(2);
191     //From Jacobi
        vec R0 = R.col(Loc[0]);
193     vec R1 = R.col(Loc[1]);
        vec R2 = R.col(Loc[2]);
195
        //Unit Test, the norm should be equal to one
197     double V0Sum=0.0, V1Sum=0.0, V2Sum=0.0;
        double R0Sum=0.0, R1Sum=0.0, R2Sum=0.0;
199
        for(int i=0; i<(n-1); i++)
201     {
            V0Sum+=V0(i)*V0(i);
203        V1Sum+=V1(i)*V1(i);
           V2Sum+=V2(i)*V2(i);
205        R0Sum+=R0(i)*R0(i);
           R1Sum+=R1(i)*R1(i);
207        R2Sum+=R2(i)*R2(i);

209     }

211     cout<<"Norm of Ground (Ar) "<<V0Sum<<endl;
              cout<<"Norm of 1st (AR) "<<V1Sum<<endl;
213           cout<<"Norm of 2nd (AR) "<<V2Sum<<endl;
              cout<<"Norm of Ground (Ja) "<<R0Sum<<endl;
215           cout<<"Norm of 1st (Ja) "<<R1Sum<<endl;
              cout<<"Norm of 2nd (Ja) "<<R2Sum<<endl;
217
```

```
219   //Writing to file , Rho, EigenFunctions
      ofile.open(outfile);
221   ofile<<setiosflags(ios::showpoint | ios::uppercase);
      ofile<<"#Rho"<<setw(23)<<"Ground(Ar)";
223   ofile<<setw(20)<<"1st(Ar)"<<setw(20)<<"2nd(Ar)";
      ofile<<setw(20)<<"Ground(Ja)"<<setw(20)<<"1st(Ja)";
225   ofile<<setw(20)<<"2nd(Ja)"<<endl;

227   for(int i=0; i<(n-1); i++)
      {
229         ofile<<rho_i[i+1];
          ofile<<setw(20)<<(1/h)*V0(i)*V0(i);
231       ofile<<setw(20)<<(1/h)*V1(i)*V1(i);
          ofile<<setw(20)<<(1/h)*V2(i)*V2(i);
233       ofile<<setw(20)<<(1/h)*R0(i)*R0(i);
          ofile<<setw(20)<<(1/h)*R1(i)*R1(i);
235       ofile<<setw(20)<<(1/h)*R2(i)*R2(i)<<endl;
      }
237
      ofile.close();
239
      //Wrting to file with Eigenvalues
241
      ofile.open(Eigen);
243   ofile<<setiosflags(ios::showpoint | ios::uppercase);
      ofile<<"#Eig(Ja)"<<setw(25)<<"Time(Ja)";
245   ofile<<setw(25)<<"iterations"<<setw(25)<<"Eig(Ar)";
      ofile<<setw(25)<<"Time(Ar)"<<endl;
247
      for (int i=0; i<5; i++)
249   {
        ofile<<setprecision(6)<<Max(i);
251     ofile<<setw(25)<<setprecision(6)<<time_j;
        ofile<<setw(25)<<iterations;
253     ofile<<setw(25)<<eigval(i);
        ofile<<setw(25)<<time_ar<<endl;
255   }


257
      ofile.close();
259
      delete [] V_Pot;
261   delete [] V_Pin;
      delete [] rho_i;
263
}//End Main Program
265


267 void rotate(mat &A, mat &R, int k, int l, int n)
    {
269
      //Rotation of the Matrix
271         double tau;
            double t;
273         double s;
            double c;
```

```
275
        if(A(k,l) != 0.0)
277     {
          tau = (A(l, l)-A(k,k))/(2*A(k,l));
279
          if(tau > 0)
281       {
            t = -tau + sqrt(1 + tau*tau);
283       }
          else
285       {
            t = -tau - sqrt(1+ tau*tau);
287       }
          c = 1/sqrt(1 + t*t);
289       s = t*c;
        }
291     else
        {
293       c =1.0;
          s= 0.0;
295   }

297   double a_kk, a_ll, a_ik, a_il, r_ik, r_il;
        a_kk = A(k,k);
299     a_ll = A(l,l);

301     //Changing the matrix elemensts with indeces k an l
        A(k, k) = c*c*a_kk - 2.0*c*s*A(k,l) + s*s*a_ll;
303     A(l, l) = s*s*a_kk + 2.0*c*s*A(k,l) + c*c*a_ll;
        A(k,l) = 0.0;
305     A(l,k) = 0.0;

307     //Changing the remaining elements
        for(int i= 0; i< (n-1); i++)
309     {
          if( i != k && i != l)
311       {
            a_ik = A(i,k);
313         a_il = A(i,l);
            A(i,k) = a_ik*c - a_il*s;
315         A(k,i) = A(i,k);
            A(i,l) = a_il*c + a_ik*s;
317         A(l,i) = A(i,l);
          }
319
          //Compute the new eigenvectors
321       r_ik = R(i,k);
          r_il = R(i,l);
323       R(i,k) = c*r_ik - s*r_il;
          R(i,l) = c*r_il + s*r_ik;
325   }
    return;
327 }
```