Schrodinger's Equation for Two Electrons in a Three-dimensional Harmonic Oscillator Well

Bryan Isherwood

March 5, 2016

Abstract

In this project we numerically solve Schrodinger's equation for two electrons within a harmonic oscillator trap. To do this we divide the project into two sections: Non-interactions and interactions. We begin by solving the non-interacting case by determining the energy eigenvalues to four leading digits of accuracy as well as the corresponding eigenvectors. By non-dimensionalizing Schrodinger's equation, we then use our previous solution to determine the interacting case by making slight changes to our potential V(ρ). Our simulation shows that the electrostatic potential acts to force the probability distribution outwards and the oscillator potential seeks to pull it back inwards.

1 Introduction

Schrodinger's equation describe the evolution of quantum mechanical systems and in its time-independent form looks like:

$$-\frac{\hbar^2}{2m}\nabla^2\Psi_\lambda(\mathbf{r}) + V(\mathbf{r})\Psi_\lambda(\mathbf{r}) = E_\lambda\Psi_\lambda(\mathbf{r}) \tag{1}$$

where V is the potential energy of the system and E_λ is the energy eigenvalue of the energy eigenfunction Ψ_λ. While one dimensional cases of [1] can be solved with relative ease, increasing the number of particles or dimensions can often create complications in finding a solution. Thus, in order to solve many quantum mechanical systems we require a numerical solution to (1). For the purpose of this project we aim to solve the harmonic oscillator problem for two identical charged particles, in our case electrons.

In order to do this, the task of computation has been broken down into two steps: non-interacting and interacting. The non-interacting case of the harmonic oscillator is solved in numerous places as each electron will act as if they are the only one's within the harmonic oscillator potential. As a result we can analytically solve for both the eigenfunctions and eigenenergies of our system. We use this analytic solution as a benchmark for our numerical solution, where we seek accuracy in our eigenenergies of up to four leading digits. This is achieved using a Jacobi Rotation algorithm that will diagonalize the approximate matrix form of (1). Once this algorithm has been optimized to our desired accuracy, we can change our effective potential to include an interaction term between the two electrons.

This paper will be structured as follows: First we will briefly discuss the theory behind the harmonic oscillator potential and how eq. (1) will be used in the algorithm. This section will be broken up into two part: Dimensionless differential equation manipulations and linear algebra manipulation. Next the structure of the algorithm itself will be discussed with a simple example that can be worked by hand. The following two sections discuss the results of the program as a whole and will make comparisons between the two cases.

2 Theory

2.1 Schrodinger's Dimensionless Equation

Our task is to determine the energy eigenvalues of the three dimensional Schrodinger equation for the harmonic oscillator potential. To this end, we first need to transform our equation into a more manageable form. We assume all coordinates have a separable relationship and convert eq. (1) into spherical coordinates:

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

This switch to spherical coordinates allows us to recast eq. (1) into a one dimensional problem. Due to the radial component of the position vector containing all the energy information we need, the azimuth and polar coordinates are not necessary for computation. Allowing the substitution $R(r)=\frac{u(r)}{r}$eq. (2) becomes:

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\left(V(r)+\frac{\hbar^2}{2m}\frac{l(l+1)}{r^2}\right)u(r)=Eu(r) \tag{3}$$

we will ignore the angular momentum number l giving:

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+V(r)u(r)=Eu(r) \tag{4}$$

Next we will switch to a dimensionless independent variable and recast the equation such that it is completely independent of any quantities of physical significance. This process is straightforward with a bit of algebra and for further details please refer to [1], for the moment there are a few quantities that are important to recognize. In the non-interacting case:

$$\begin{aligned}\rho&=\frac{r}{\alpha}\\ V(\rho)&=\frac{1}{2}k\alpha^2\rho^2\\ \alpha&=\left(\frac{\hbar^2}{mk}\right)^{1/4}\\ \lambda&=\frac{2m\alpha^2}{\hbar^2}E\end{aligned} \tag{5}$$

rewriting eq. (4) to:

$$-\frac{d^2}{d\rho^2}u(\rho)+\rho^2u(\rho)=\lambda u(\rho) \tag{6}$$

The value λ is now proportional to the energy eigenvalue for our specific eigenstate u. As a result, the program attached at the bottom gives back λ as eigenvalues. In the interacting case we need to include the electrostatic potential of the two electrons:

$$\begin{aligned} \alpha &= \frac{\hbar^2}{m\beta e^2} \\ \omega_r^2 &= \frac{mk}{4\hbar^2}\alpha^4 \\ V(\rho) &= \frac{1}{2}k\alpha^2\rho^2 \\ \lambda &= \frac{m\alpha^2}{\hbar^2}E \end{aligned} \tag{7}$$

where β is the proportionality constant of the electrostatic potential and ω_r is the dimensionless angular frequency of the oscillator. Using these substitutions we can write our interaction equation as:

$$-\frac{d^2}{d\rho^2}u(\rho) + \Big(\omega_r^2\rho^2 + \frac{1}{\rho}\Big)u(\rho) = \lambda u(\rho) \tag{8}$$

Looking at equations (6) and (8) it can be seen that we inevitably created two new effective potentials, call them *V'*, such that:

$$\begin{aligned} V'_{nint} &= \rho^2 \\ V'_{int} &= \omega_r^2\rho^2 + \frac{1}{\rho} \end{aligned} \tag{9}$$

for the non-interacting and interacting cases, respectively. We are now ready to break down our equations into a more computationally friendly formalism.

2.2 Schrodinger's Discrete Matrix Equation

As it has been shown in [2], we can decompose our second derivative into an approximate form that will allow us to exchange our PDE for a linear algebra equation. Discretizing our second derivative gives us a matrix of the form:

$$\frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \dots & -1 & 2 & -1 \\ 0 & 0 & \dots & \dots & -1 & 2 \end{bmatrix}$$

where we will define $h = \frac{\rho_{max}-\rho_{min}}{n+1}$, where n is the number of discrete sections into which we have broken our functional domain. The potential can be broken down in a similar fashion to make:

$$A = \begin{bmatrix} \frac{2}{h^2}+V'_1 & \frac{-1}{h^2} & 0 & 0 & \dots & 0 \\ \frac{-1}{h^2} & \frac{2}{h^2}+V'_2 & \frac{-1}{h^2} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \dots & \frac{-1}{h^2} & \frac{2}{h^2}+V'_{n-2} & \frac{-1}{h^2} \\ 0 & 0 & \dots & \dots & \frac{-1}{h^2} & \frac{2}{h^2}+V'_{n-1} \end{bmatrix}$$

where $V_i' = V'(ih)$. Our equation now reduces too:

$$A\mathbf{u}_\lambda = \lambda\mathbf{u}_\lambda \tag{10}$$

In order to determine our eigenvalues, we preform a series of rotations on A using the rotation matrix S:

$$S = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & \cos\theta & 0 & \dots & 0 & \sin\theta \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 1 & \dots \\ 0 & 0 & \dots & -\sin\theta & 0 & \dots & 0 & \cos\theta \end{bmatrix}$$

with the property:

$$S^T = S^{-1} \tag{11}$$

we then preform a series of these rotations on eq. 10:

$$(SAS^T)S\mathbf{u}_\lambda = BS\mathbf{u}_\lambda = \lambda S\mathbf{u}_\lambda \tag{12}$$

leading to:

$$A'\Big(\prod_n S_n\mathbf{u}_\lambda\Big) = \lambda\Big(\prod_n S_n\mathbf{u}_\lambda\Big) = \lambda R\mathbf{u}_\lambda \tag{13}$$

where *A'* is the diagonal matrix whose diagonal elements are the eigenvalues of A and R is the matrix whose columns are the eigenvectors of A. This process is known as a Jacobi Rotation [1]. The guiding principle behind the Jacobi rotation is similar to that of determining the principle axes of a rotating body. While we are given a set of super positions of our wavefunction *u(x)*, we can rotate our discrete hamiltonian until we reach a point where each value for our wavefunction lies along its own axis. In the next section we will discuss the algorithm for implementing this rotation.

3 Algorithm

The basic algorithm used for this project was the Jacobi Rotation method of determining matrix eigenvalues and eigenvectors. A pre-written householder algorithm was used to test the outcome of the rotations and the steps for this method will not be discussed in this paper. To accomplish and simplify our rotation it is faster to redefine each element of the given matrix rather than simply perform a series of matrix multiplications. For comparison, multiplication of two n *x* n matrices costs $\approx n^3$ flops per rotation, where as reassignment costs $\approx n^2$ flops per rotation.

First, we need to determine the angle over which we rotate our matrix elements. In order to do this, we require knowledge of the largest elements in our matrix. Creating a temporary variable and scanning each of the matrix elements until there are no larger elements (in magnitude) quickly solves our first problem. For example:

$$if(max < a_{kl})max = a_{kl}; for\ l \neq k \tag{14}$$

The inequality is important as we do not want to cause the diagonal elements to become zero in the following steps. Using our new largest value we can calculate the sine and cosine of our angle after several quick steps:

$$\begin{aligned} cot(2\theta) &= \tau = \frac{a_{ll} - a_{kk}}{2a_{kl}} \\ tan(\theta) &= t = -\tau \pm \sqrt{1+\tau^2} \\ cos(\theta) &= c = \frac{1}{\sqrt{1+t^2}} \\ sin(\theta) &= s = cos(\theta)tan(\theta) = ct \end{aligned} \tag{15}$$

When determining the value of t, it is important to use the radical that gives the smallest value of $tan\theta$ [1]. This will guarantee that $|\theta| < \frac{\pi}{4}$, a necessary condition to ensure that t does not become too large. As an example of the process, suppose we had a matrix of the form:

$$A = \begin{bmatrix} 5 & -1 \\ -1 & 4 \end{bmatrix} \tag{16}$$

The maximum off diagonal is -1, giving values of $tan(\theta) = .618$, $cos(\theta) = .851$, $sin(\theta) = .526$. In the next step we apply our rotation to each element:

$$\begin{aligned} &a'_{kl} = (a_{kk} - a_{ll})cs + a_{kl}(c^2 - s^2) = 0 \\ &a'_{kk} = c^2 a_{kk} - 2csa_{kl} + s^2 a_{ll} \\ &a'_{ll} = s^2 a_{kk} + 2csa_{kl} + c^2 a_{ll} \\ &if(i \neq k, j \neq l) \\ &\quad a'_{ik} = ca_{ik} - sa_{il} \ ; \ a'_{il} = ca_{il} + sa_{ik} \\ &\quad a'_{ii} = a_{ii} \end{aligned} \tag{17}$$

where the primed coordinates define our matrix elements after the rotation. There are two points of note here: First, for our purposes the resulting matrix *A'* is symmetric and all off diagonal replacements have to occur twice such that $a'_{ij} = a'_{ji}$. Second, the first line in eq. (5) is trivially zero by trigonometric identities, as is the purpose of the algorithm. To further account for this, the actual algorithm should just define $a'_{kl} = 0$ to save computational time. Applying this algorithm to our matrix A gives eigenvalues of **3.382** and **5.618**, when using the attached program. In order to acquire the eigenvectors for the given matrix we simply need to act the following substitutions on the "normal basis" or the identity matrix:

$$\begin{aligned} r'_{ik} &= cr_{ik} - sr_{il} \\ r'_{il} &= cr_{il} + sr_{ik} \end{aligned} \tag{18}$$

where r_{ij} is an element of the matrix that stores our eigenvectors. For the given example, the resulting eigenvector matrix looks like:

$$R = \begin{bmatrix} .851 & -.526 \\ .526 & .851 \end{bmatrix} \tag{19}$$

If the columns are read as vectors, it is clear to see that the dot product of these vectors will result with *0*, therefore orthogonality is preserved, as expected for eigenvectors.

In the implementation of this algorithm a bubble sorter was used to arrange the eigenvalues from smallest to largest and to rearrange their corresponding eigenvectors. While the i^{th} eigenvalue in our newly diagonalized matrix will always correspond to the i^{th} column in $\mathbf{R}$, the values and vectors will not automatically be in order, thus for ease of reading a light sorting algorithm of some kind is suggested.

4 Results

The method described above was implemented by the program **project2.cpp** using the package **Lib.cpp**. While **Lib.cpp** allowed the use of a householder algorithm, it was primarily used to check the results of the Jacobi Rotation algorithm and will not be discussed here. The program **project2.cpp** determines the eigenvalues and eigenvectors of our given discrete hamiltonian matrix A. By scanning over multiple values of n, it was found that the number of Jacobi rotations goes as:

$$N(n) \approx 1.7664n^2 - 10.215n \tag{20}$$

Eq. (20) will over or under approximate depending on the value of n, although generally speaking it will over approximate for large vales in n and under approximate for small values of n with an average error of < 10 percent. Values used to determine $N(n)$ can be found in **benchmark.txt**.

Figure 1 shows shows the results from the first three eigenstates of the non-interacting harmonic oscillator, corresponding to λ = 3,7,11. Figure 1 clearly shows the three eigenstates one might expect from the harmonic oscillator, each having K-1 nodes that correlate to the $\mathrm{K^{th}}$ eigenstate. The plot was generated using n = 299 and generated eigenvalues of $\lambda = 3.00156, 7.00036, 10.9987$, requiring 154,926 Jacobi Rotations to reduce the non-diagonal elements of A to zero.

One point of note is that implementation of this code for different values of n will result in different normalization constants and therefore peak values. This is due to the discrete nature of our calculation compared to its typical continuous calculation, for every added division we are not only adding a new layer of accuracy to the calculation but also picking a new point off of the actual analytic function. Doing this causes the normalization constant to increase with the number of divisions, while the peak of the function may not increase proportionately. As a result the figures shown here should not be considered numerically significance, but rather as a display of qualitative differences in the spacial distribution of probabilities.

To implement the interacting case the electrostatic coulomb potential was added and the results of this calculation can be seen in the appendix for values of $\omega_r = 0.01, 0.5, 1.0, 5.0$. Figure 2 shows a comparison of the ground states of the non-interacting case as well as the interacting case for all ω_r, to see the first and second excited states for the interacting case, please view figures 3- 6 in the appendix. The program will also implement a unit test by checking that each of the eigenvalues is orthonormal. To do this, the program take the dot product of the ground state eigenvector as well as the $(n-1)^{st}$ state. While the actual value computed will vary by system parameters, the typical dot product was < 0.001 and was independent of the eigenvectors used.

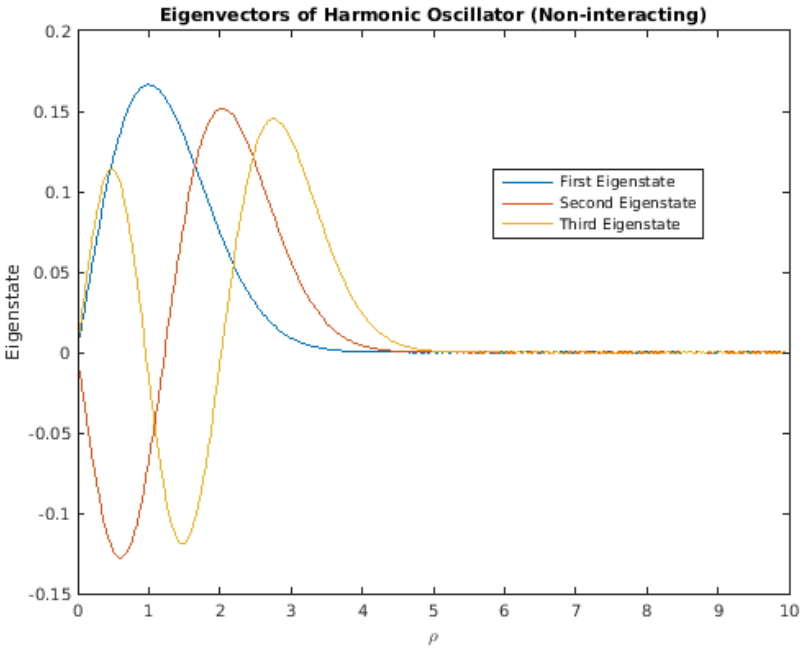

Figure 1: This plot shows the three lowest eigenstates for the 3D non-interacting harmonic oscillator potential as a function of ρ. The domain covers 299 divisions with a max radial distance of 10. This figure was made using matlab plotting software.

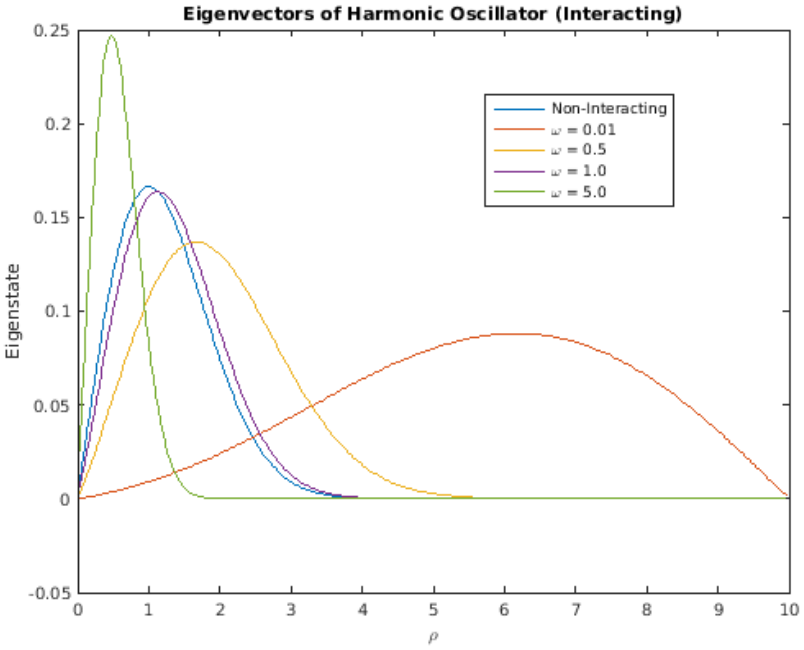

Figure 2: This plot shows the ground states for the 3D harmonic oscillator potential with and without coulomb interaction as a function of ρ. The domain covers 299 divisions with a max radial distance of 10. This figure was made using matlab plotting software.

5 Analysis

Figure 2 shows how the ground state eigenvector changes by initial parameters, the non-interacting case in blue will act as our benchmark state. As ω_r increased, the height of the peak increases and the total width of the distribution narrows. This is likely a result of the coulomb potential overtaking the harmonic oscillator potential for small values of ω_r. Coulomb's force law tells us that two alike charges will repel one another and as a result the average position of the electrons moves outwards as the harmonic oscillator is unable to contain the electrostatic force. Even without angular momentum, the electrons still manage a slow decay in probability towards the origin due to these forces.

Increasing ω_r, or equivalently the elastic constant of the harmonic oscillator, causes the distribution function to narrow as the electrostatic potential can no longer overcome oscillator. Classical mechanics can actually shed some light onto this notion. Mechanics teaches us that any driven oscillation of sufficient strength will eventually overcome any other constantly acting forces. Eq. (6) reminds us that the oscillator frequency is proportional to the strength of the harmonic oscillator, thus as ω_r increases so does the strength of the oscillator. This force overcomes the electrostatic force that is repelling the electrons and collects them in a very narrow region.

If we compare our energy eigenvalues (see appendix for details) we gain more insight into the interaction. The electrostatic force seeks to minimize the energy eigenvalues, as it is attempting to move the electrons as far away from each other as possible. As ω_r increases the energy eigenvalues increase as well, creating a strong interaction energy as the oscillator forces the electrons to exist in close proximity.

6 Conclusion

The goal of this project was to use successive Jacobi Rotations to determine the energy eigenvalues and vectors of the quantum harmonic oscillator with and without including electrostatic interactions between two electrons. Our program showed that Jacobi Rotations were easily capable of determining these values in a sufficiently fast manner. Comparing our two cases, we could see that the effect of the electrostatic potential was to force the electrons away from their center of mass position, while the harmonic oscillator attempted to pull them back. As the strength of the oscillator increased, the average deviation away from the center of mass position decreased, showing its capability to overcome the electrostatic potential.

7 References

[1] M. Hjort-Jensen. Computational physics, lecture notes spring 2016. Department of Physics, Michigan State University, 2016.

[2] B. Isherwood. *Evaluation of Poissions Equation by Numerical Approximation.* Department of Physics, Michigan State University, 2016.

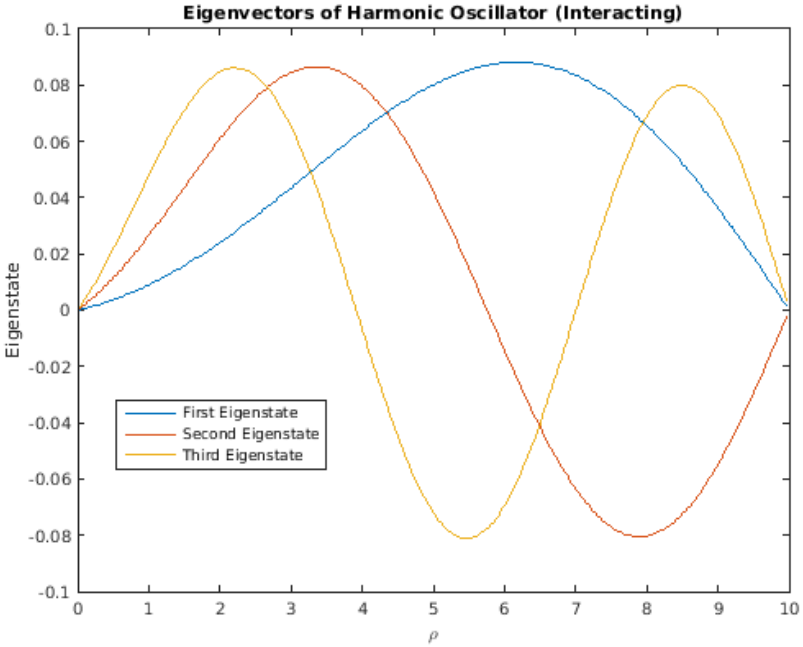

Figure 3: This plot shows the three lowest eigenstates for the 3D interacting harmonic oscillator with $\omega_r = 0.01$ potential as a function of ρ. The domain covers 299 divisions with a max radial distance of 10. This figure was made using matlab plotting software. $\lambda = 0.314, 0.683, 1.224$

8 Appendix

```
#include <iostream>
#include <string>
#include <cmath>
#include <fstream>
#include <cstdlib>
#include <new>
#include "lib.h"
#include "time.h"

using namespace std;



int main()
{
    //double *p;
    double **A,**V, **R;
    int i,j,k,l,n,jt,it, count = 0;
    double akl,akk,all,aik,ail,rik,ril,tempu,tempb,max = 1.0, t,tau,c,s;
    double rmin = 0.0,rmax,h,h2,c2,s2,omeg,omeg2,on,off,r0,r1,r2;
```

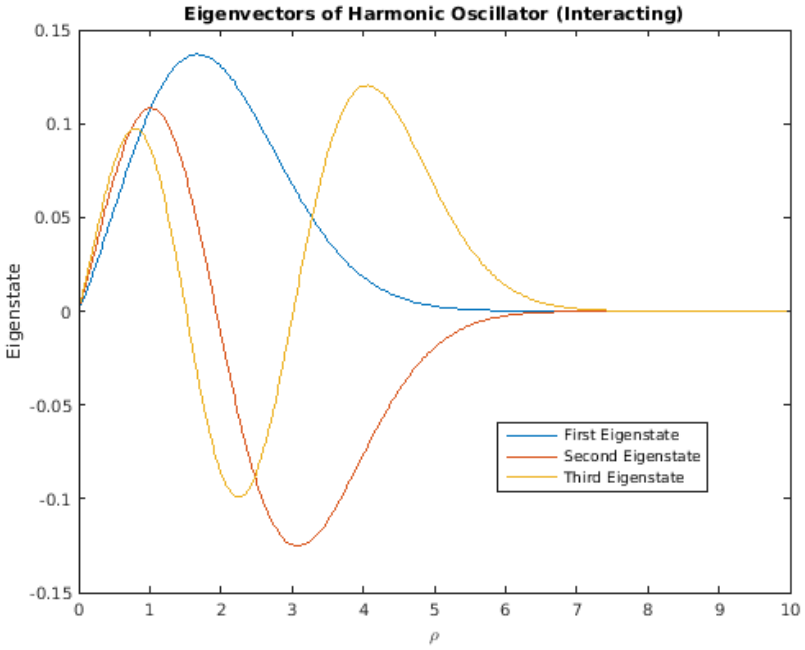

Figure 4: This plot shows the three lowest eigenstates for the 3D interacting harmonic oscillator with $\omega_r = 0.5$ potential as a function of ρ. The domain covers 299 divisions with a max radial distance of 10. This figure was made using matlab plotting software. $\lambda = 2.231, 4.136, 6.074$

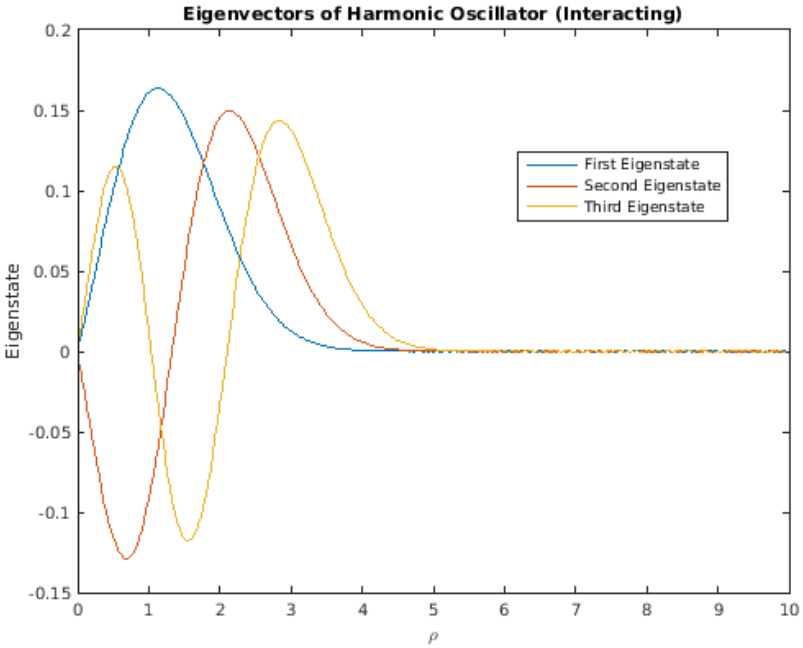

Figure 5: This plot shows the three lowest eigenstates for the 3D interacting harmonic oscillator with $\omega_r = 1.0$ potential as a function of ρ. The domain covers 299 divisions with a max radial distance of 10. This figure was made using matlab plotting software. $\lambda = 4.059, 7.9096, 11.816$

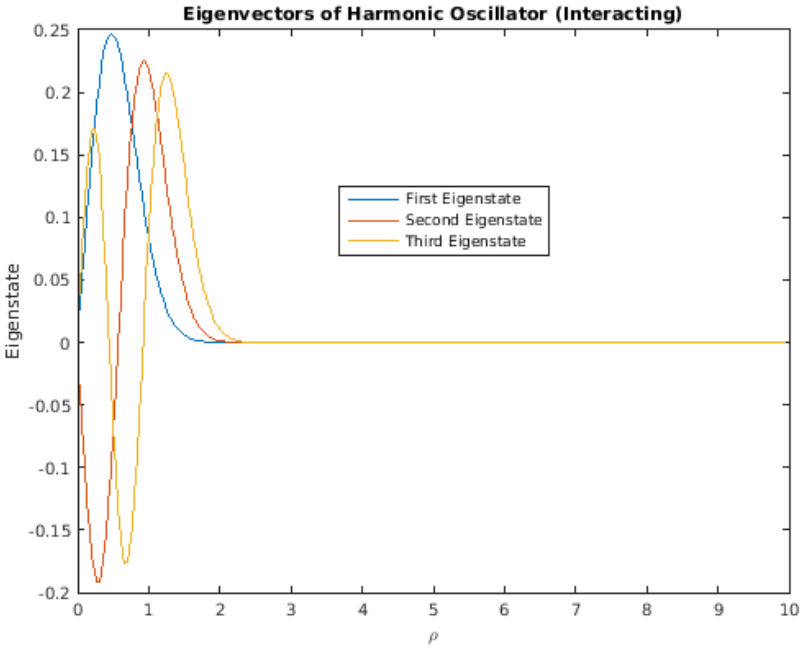

Figure 6: This plot shows the three lowest eigenstates for the 3D interacting harmonic oscillator with $\omega_r = 5.0$ potential as a function of ρ. The domain covers 299 divisions with a max radial distance of 10. This figure was made using matlab plotting software. $\lambda = 17.441, 37.032, 56.750$

```
double *B,*e,*U;
string answer;
cout<< "Please give a stepsize" << endl;
cin >> n ;
cout << "Please give a max radial distance" << endl;
cin << rmax;
cout << "Would you like to turn on electrostatic forces?" << "YES |
    NO" << endl;
cin >> answer;
if(answer == "yes" || answer == "Yes" || answer == "YES"){
    cout << "Please enter an oscillation frequency" << endl;
    cin >> omeg;
}

omeg2 = omeg*omeg;




        U = new double[n];
        e = new double[n];
        B = new double[n];
        A = (double **) matrix(n,n,sizeof(double));
        V = (double **) matrix(n,n,sizeof(double));
        R = (double **) matrix(n,n,sizeof(double));



h = (rmax-rmin)/(n+1);
h2 = h*h;
on = 2.0/h2;
off = -1.0/h2;




for (i=0; i<n; i++){
    for (k=0; k<n; k++){
        if(i == k){
            if(answer == "yes" || answer == "Yes" || answer == "YES"){
            A[i][k] = on+omeg2*(i+1.0)*(i+1.0)*h2 - 1.0/((i+1)*h);}
            else{
                A[i][k] = on+(i+1.0)*(i+1.0)*h2;
            }
          //  V[i][k] = (i+1.0)*(i+1.0)*h2;
            R[i][k] = 1.0;
            B[i] = A[i][k];
            e[i] = off;}
        else if (k == i-1){
            A[i][k] = off;
            R[i][k] = 0.0;}
        else if (k == i+1){
            A[i][k] = off;
            R[i][k] = 0.0;}
```

```
            else{
                A[i][k] = 0.0;
                R[i][k] = 0.0;}


       //  A[i][k] = A[i][k] + V[i][k];

        }
       }
    free_matrix((void **) V);


    //tqli(B,e,n,R);


  //  for (i = 0; i<n; i++){
    //    cout<< A[i][0] << " "<<A[i][1]<<" "<<A[i][2] <<endl;}

   while(fabs(max) >= 0.0000000001){
   // for(k = 0; k<2;k++){
       max = 0.0;

    for(i = 0; i<n;i++){
        for(j = i+1; j<n; j++)
            if(fabs(A[i][j]) >= fabs(max)){
                max = A[i][j]; it = i; jt = j; }}

akl = 2.0*A[it][jt];
    if (A[it][jt] != 0){
        tau = (A[jt][jt]-A[it][it])/akl;
    if(tau >= 0){
        t = -tau + sqrt(1+tau*tau);}
    else{
        t = -tau - sqrt(1+tau*tau);}


    c = 1/sqrt(1+t*t);
    s = c*t;
    //cout << max<< " " << fabs(max) << endl;
    }
    else{
        c = 1.0;
        s = 0.0;
    }


    c2 = c*c;
    s2 = s*s;
    akk = A[it][it];
    all = A[jt][jt];
    A[it][it] = c2*akk-c*s*akl+s2*all;
    A[jt][jt] = s2*akk + c*s*akl+c2*all;
    A[it][jt] = 0.0;
    A[jt][it] = 0.0;
```

```
    for(i = 0;i<n;i++){
        if(i != it && i != jt){
            aik = A[i][it];
            ail = A[i][jt];
            A[i][it] = c*aik -s*ail;
            A[it][i] = A[i][it];
            A[i][jt] = c*ail + s*aik;
            A[jt][i] = A[i][jt];
            //cout << aik << endl;

        }
 rik = R[i][it];
 ril = R[i][jt];

 R[i][it] = c*rik - s*ril;
 R[i][jt] = c*ril + s*rik;
}

   // cout<< fabs(max)<< " " << it << " " << jt <<endl;
    count += 1;
}
```

```
   // akl = off_diag(A,n); //returns pointer array whose elements are
       the largest off diagonal and its respective coordinates
    //it = p[1];
   // jt = p[2];

for (i = 0; i<n; i++){
    U[i] = A[i][i];
   // cout << R[i][39] << " " << i << endl;}
}
```

```
//Sort Eigenvalues and vectors
    for (i = 0; i<n; i++){
        for(j=0;j<n-1;j++){
            if(U[j]>U[j+1]){
                tempu = U[j+1];
                U[j+1] = U[j];
                U[j] = tempu;
                for(k=0;k<n;k++){
                 tempu = R[k][j];
                  R[k][j] = R[k][j+1];
                  R[k][j+1] = tempu;
```

```
            }
        }
        if(B[j]>B[j+1]){
            tempb = B[j+1];
            B[j+1] = B[j];
            B[j] = tempb;
        }

    }}
// cout<< B[0] << " " << B[1] << " " << B[2] << endl;
// cout << A[i][i]<< endl;


 ofstream myfile;
    myfile.open("proj2d.txt");

    for( i = 0; i<n; i++){
        r0 += R[i][0]*R[i][0];
        r1 += R[i][1]*R[i][1];
        r2 += R[i][2]*R[i][2];

    }
   /* for(i=0; i< n ; i++){

        myfile << R[i][0]/sqrt(r0) << " "<<R[i][1]/sqrt(r1) <<" "
            <<R[i][2]/sqrt(r2) << "\n";



 }*/

for (i = 0;i<n;i++){
    tempu += R[i][0]*R[i][n-1];
}



cout<< "n = " << n << " Rmax = "<< rmax << endl;
cout << "Groundstate eigenvalue is " <<U[0]<< endl;
cout << "First excited state eigenvalue is " << U[1] << endl;
cout << "Second excited state eigenvalue is " << U[2] << endl;
cout << "This calculation required " << count - 1 << " Jacobi Rotations"
    << endl;

delete [] U;
delete [] B;
delete [] e;
free_matrix((void **) R);
free_matrix((void **) A);
```

```
    return 0;

}
```