PHY 480 - Computational Physics
Project 2: Schrödinger's Equation for 2 electrons in a 3D Well

Thomas Bolden

March 4, 2016

Github Repository at https://github.com/ThomasBolden/PHY-480-Spring-2016

Abstract

In this project, a solution was found to the two-electron Schrödinger equation in three-dimensions. This was done by diagonalizing an n by n matrix to find the eigenfunctions and eigenvectors of the three lowest energy states. Computation time to determine the eigenvalues was compared between a brute-force implimentation of Jacobi's algorithm and the Armadillo library. For large n, the Armadillo library is orders of magnitude faster. Using the Armadillo *eig_sys* function, the wavefunctions of the lowest three energies are compared while varying the strength of the harmonic oscillator potential.

Contents

Introduction

In physics, there is a known solution to Schrödinger's equation for a single electron in a spherically symmetric three-dimensional well. However, this equation becomes more complicated when another elctron is added. In addition to the energy term, there are Coulombic interactions between the electrons that must be accounted for.

For one electron, the radial part of the Schrödinger equation reads

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

In this case, the potential function for the harmonic oscillator is $V(r)=\dfrac{m\omega^2}{2}$, and E is the 3D energy. With th quantum numbers n and l, the energies are given by

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

If we are to solve this computationally, it is helpful to introduce a dimensionless variable $\rho=1/\alpha$, with α is the length constant.

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

If we neglect angular momentum by setting $l=0$, and making $\omega=\alpha\rho$, we get

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

If both sides of thie equation are multiplied by $\dfrac{2m\alpha^2}{\hbar^2}$ and fixing $\alpha=\left(\dfrac{\hbar^2}{mk}\right)^{1/4}$ the equation becomes

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

Discrete energy eigenvalues are given by

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

The first three eigenvalues are the $\lambda_0=3$, $\lambda_1=7$, $\lambda_2=11$.

In order to solve this, the standard approximation of the second derivatve with step length h,

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

will be used. The finial tridiagonal matrix is then

$$\begin{pmatrix}
\frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0\\
-\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0\\
0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0\\
\dots & \dots & \dots & \dots & \dots & \dots & \dots\\
0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{n_{\text{steps}}-2} & -\frac{1}{h^2}\\
0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{n_{\text{steps}}-1}
\end{pmatrix}.$$

In this project, I explore and compare various ways to solve the two-electron Schrödinger equation. In the following section, the methods used to write and impliment the Jacobi rotation code are discussed. In the Results section, I present the data I have found. I discuss it in some detail, but most of the analysis is saved for the Conclusions section. Finally, the Code section is a collection of the code used to produce the results.

Methods

The general algorithm for Jacobi's rotation is as follows:

Update !! !!!!!!!!!!!!!!!!!!!!!!!!!!!!

Algorithm 1 Jacobi Rotation

1: **procedure** J(a)cobi Rotation
2: $n \times n$ matrix A
3: **function** M(a)xOffDiagonal
4: **end function**
5: **end procedure**

When implimenting this algorithm, it is important to choose the smaller magnitude solution to $t = -\tau \pm \sqrt{1+\tau^2}$. To get a sense of why, it is best to use the equation

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

This equation, with $c = \dfrac{1}{1+t^2}$, gives the difference between matrices **A** and **B**. Since we want to minimize this difference, it becomes clear that we want to take a value of c to be as close to 1 as possible. Doing this will make the $(1-c)$ term go to zero. Since we want a value of c that is close to one, the t value chosen must be as small as possible. This is because of the way c is defined above. With small t, the denominator approaches 1, allowing c to also approach 1.

Results

When the dimensionality of the matrix is changed, the dimensionless variable ρ needed to accurately determine the lowest three eigenvectors also changes. The data is graphed in Table 1, and graphed in Figure 1.

Table 1: Checking ρ Dependency for Various Matrix Dimensionality

Minimum $\rho_{\max}$	Matrix Dimensions n
5	200
6	250
7	300
8	350
9	400
10	450

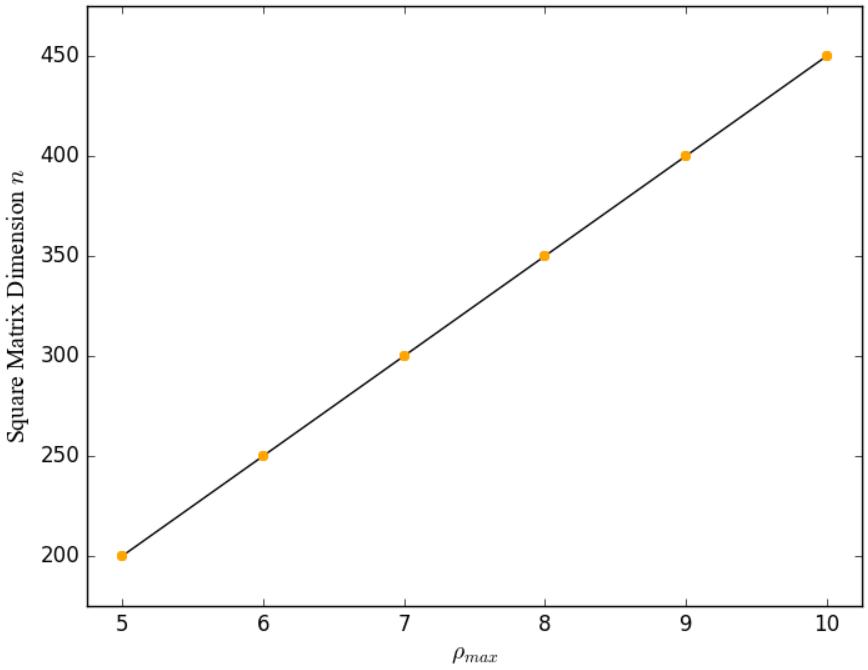

Figure 1: Value of ρ Required for Accurate Eigenvalues

With a step size of 50 for the matrix dimensinality, the relationship between ρ and n is perfectly linear.

Using this fact, one can test the number of similarity transformations required to effectively diagonalize the matrices. Implimentation of `EigensolverTests.cpp` gives these results. Here, the tolerance for a nondiagonal matrix element was set to $\epsilon = 1 \times 10^{-10}$. The number of transformations required to diagonalize a square matrix of dimension n are listed in Table 2. The computation times are also compared between Jacobi's algorithm and Armadillo's library.

Table 2: Comparing Jacobi's Algorithm to Armadillo Using $\rho = 5.0$

Dimensionality n	Eigenvalues	$t_{\text{Armadillo}}$ (s)	t_{Jacobi} (s)	Transformations
50	$(2.99687, 6.98434, 10.9619)$	0.000617	0.08572	4139
100	$(2.99922, 6.99609, 10.9907)$	0.003777	1.15983	17293
150	$(2.99965, 6.99827, 10.9960)$	0.007561	5.89020	39377
200	$(2.99980, 6.99903, 10.9978)$	0.012636	18.5741	70538
250	$(2.99988, 6.99938, 10.9987)$	0.019315	49.3644	110646
300	$(2.99991, 6.99957, 10.9991)$	0.032661	110.025	160133
350	$(2.99994, 6.99968, 10.9994)$	0.061280	248.396	218405
400	$(2.99995, 6.99976, 10.9996)$	0.086203	430.594	284916
450	$(2.99996, 6.99981, 10.9997)$	0.094821	901.509	361794
500	$(2.99997, 6.99985, 10.9998)$	0.123586	1394.52	447415

From this data, the number of transformations based on the dimensionality of the matrix was graphed. The curve was fit quadratically using *polyfit* from the numerical python library. The graph and its equation are shown in Figure 2.

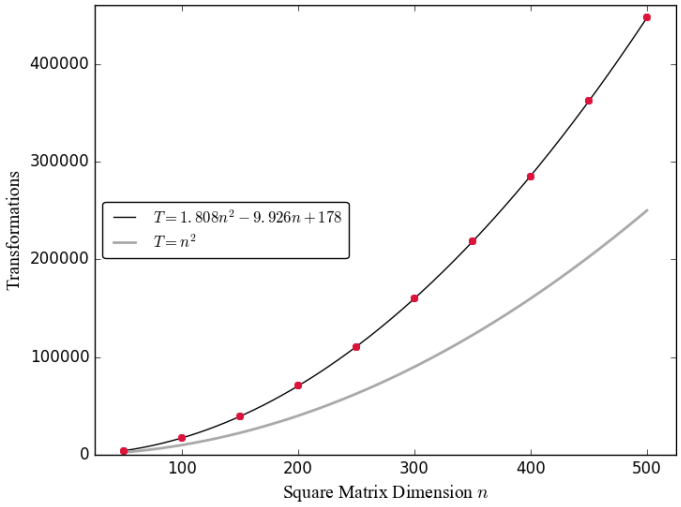

Figure 2: Tansformations Increasing with Dimensionality

The next task was to plot the two-electron wavefunctions as a function of the relative coordinate ρ and varying values of ω_r. In the figures that follow, the two-electron wavefunctions are plotted as probability distributions. Four values for ω_r were used and compared. They are: $\omega_r = \{0.01, 0.50, 1.00, 5.00\}$. The wavefunctions assumed the electron in the ground state, with $l = 0$. The first three eigenvectors graphed were found using Armadillo's *eig_sys* function.

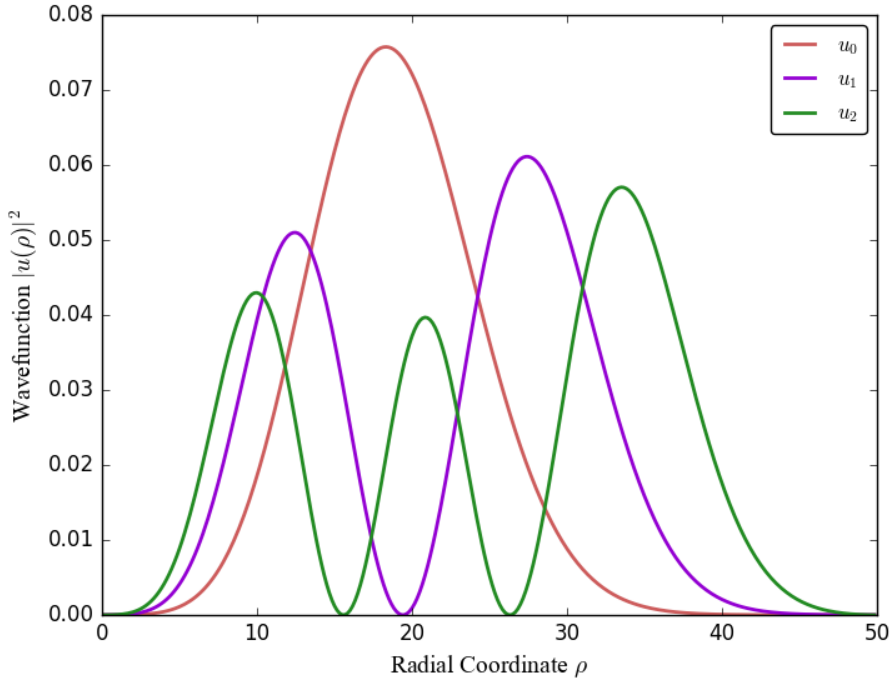

Figure 3: Wavefunction with $\omega_r = 0.01$

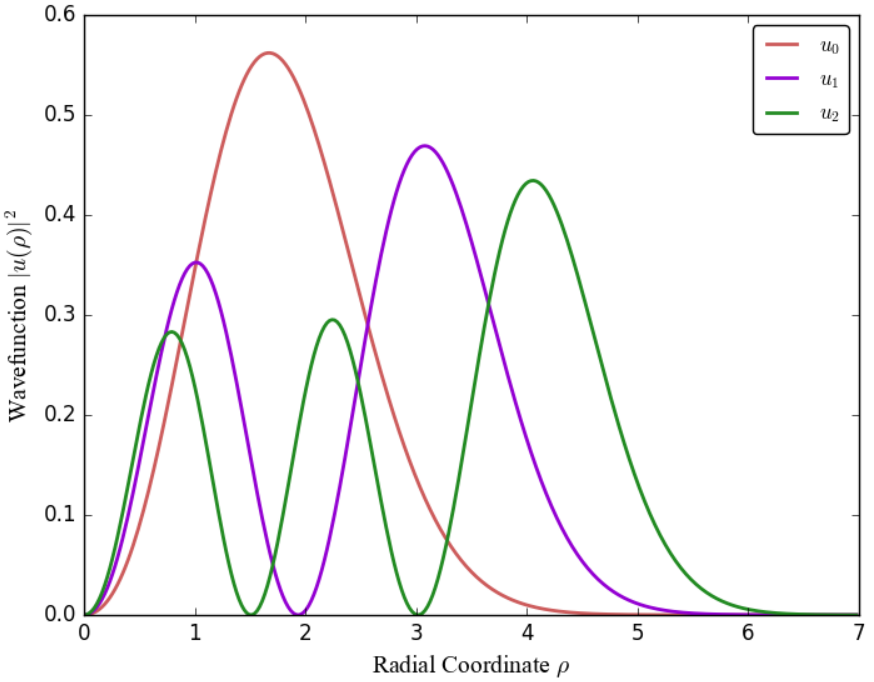

Figure 4: Wavefunction with $\omega_r = 0.50$

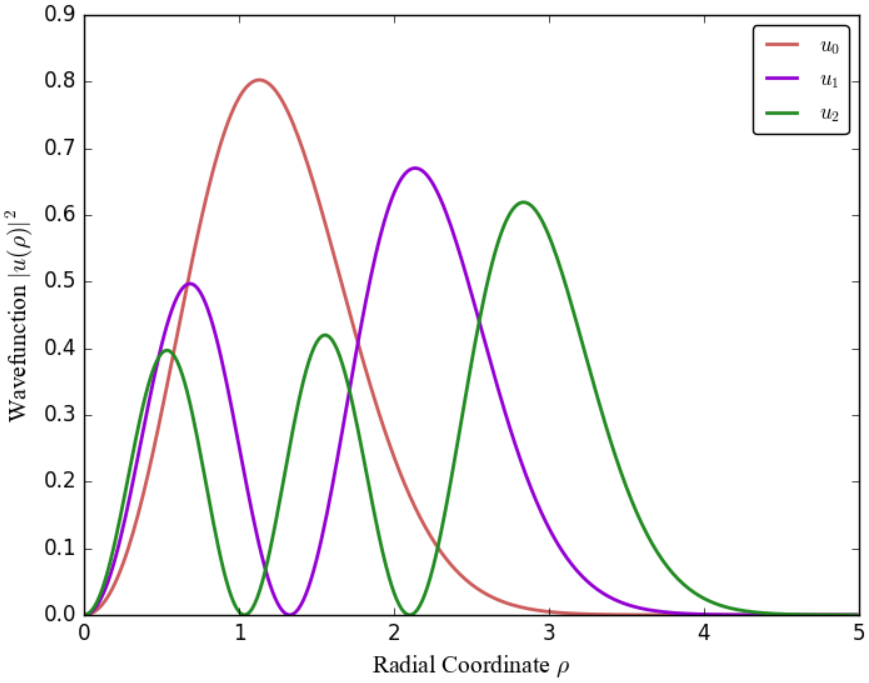

Figure 5: Wavefunction with $\omega_r = 1.00$

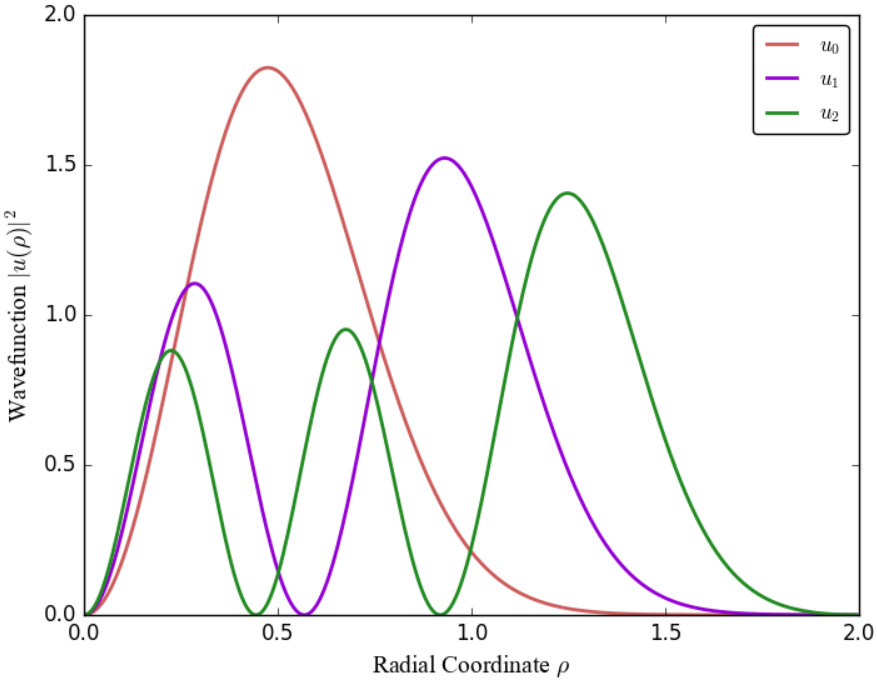

Figure 6: Wavefunction with $\omega_r = 5.00$

Conclusions

The first task was to impliment algorithms that solve a one electron system in a three dimensional harmonic oscilator potential well. To do this, a square matrix was diagonalized using both a brute-force Jacobi rotation algorithm and Armadillo. Jacobi's method for diagonalizing matrices was found to be orders of magnitude slower than the Armadillo library's *eig_sys* function. This means that, while useful and simple to impliment, Jacobi's method is only practical for small matrices.

The next task was to compare the transformations required to diagonalize a matrix based on its dimensionality. The transformations seemed to scale as nearly the square of the dimensions.

The final task was to use the most efficient and accurate settings for $\rho_{\max}$ and n, and extend it for the two-electron model. The wavefunctions were tested with varying oscillator strength (ω_r), and it was found that the relationship between ω_r and the wavefunction spread across ρ was inverse. In other words, as ω_r increased, the wavefunction compressed.

All in all, both Jacobi rotation and Armadillo are accurate methods to diagonalize matrices. However, in order to be useful, the program must also be efficient. So, if one is to use either of these methods in the future, brute-force Jacobi rotation is not fast enough, and the Armadillo library is much more reasonable.

Code

../Code/EigensolverTests.cpp

```
/*
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        Project 2 - Schrodinger's Equation for two electrons in
                    a three-dimensional harmonic oscillator well

        Part a)   - Jacobi's rotation algorithm "brute force".

        Part b)   - Compare to Armadillo to find most efficient
                    algorithm.
                    Checking for n and rho dependency.

        Results are saved to a text file.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
*/

#include <iostream>
#include <fstream>
#include <iomanip>

#include <string>
#include <time.h>

#include <cmath>
#include <armadillo>

using namespace std;
using namespace arma;
```

```
28
29 ofstream myfile;
30
31 /*
32   ======= Function to return indices of largest off-diagonal element =======
33     Arguments:
34     mat A - matrix
35     int n - size of square matrix
36     int *k - new row
37     int *l - new column
38 */
39
40 double MaxOffDiagonal(mat A, int *k, int *l, int n){
41
42     double maxOD;
43     double a_ij;
44
45     for (int i = 0; i < n; i++){
46
47         for (int j = i+1; j < n; j++){
48
49             a_ij = fabs(A(i,j));
50
51             if (a_ij > maxOD){
52
53                 maxOD = a_ij;
54                 *k = i;
55                 *l = j;
56
57             }
58         }
59     }
60     return maxOD;
61 }
62
63 /*
64   ============ Function implimenting Jacobis rotation algorithm ============
65     Arguments:
66     mat &A - reference matrix A
67     mat &B - reference matrix B
68     int k - new row index
69     int l - new column index
70     int n - dimensionality of matrix
71 */
72
73 void Rotate(mat &A, mat &B, int k, int l, int n){
74
75     double t;
76     double tau;
77     double sine;
78     double cosine;
79
80     if ( A(k,l) != 0.0){
81
```

```
82          tau = ( A(l,l) − A(k,k) )/( 2*A(k,l) );
83
84          if (tau >= 0.0) {
85
86              t = 1.0/(fabs(tau) + sqrt(1.0 + tau*tau));
87
88          }
89
90          else {
91
92              t = −1.0/(fabs(tau) + sqrt(1.0 + tau*tau));
93
94          }
95
96          cosine = 1.0/ sqrt (1.0 + t*t);
97          sine = t*cosine;
98
99      }
100
101     else {
102
103         sine = 1.0;
104         cosine = 0.0;
105
106     }
107
108     double A_ik;
109     double A_il;
110     double B_ik;
111     double B_il;
112     double A_kk = A(k,k);
113     double A_ll = A(l,l);
114
115     A(k,k) = A_kk*cosine*cosine − 2*A(k,l)*cosine*sine + A_ll*sine*sine;
116     A(l,l) = A_ll*cosine*cosine + 2*A(k,l)*cosine*sine + A_kk*sine*sine;
117     A(k,l) = 0.0;
118     A(l,k) = 0.0;
119
120     for (int i = 0; i < n; i++){
121         if (i != k && i != l){
122
123             A_ik = A(i,k);
124             A_il = A(i,l);
125             A(i,k) = A_ik*cosine − A_il*sine;
126             A(k,i) = A(i,k);
127             A(i,l) = A_il*cosine + A_ik*sine;
128             A(l,i) = A(i,l);
129
130         }
131
132         B_ik = B(i,k);
133         B_il = B(i,l);
134         B(i,k) = B_ik*cosine − B_il*sine;
135         B(i,l) = B_il*cosine + B_ik*sine;
```

```
136
137     }
138     return;
139 }
140
141 int main(){
142
143     int n;
144     int loops;
145     string filename;
146     double rho;
147     double h;
148     double e;
149     double jacobi_time;
150     double armadillo_time;
151     rowvec N;
152
153     int maxloops = 1e8;
154     double epsilon = 1e-10;
155
156     filename = "Comparisons9.txt";
157     rho = 9.0;
158
159 //  cout << "Enter a name for the file: ";
160 //  cin >> filename;
161 //  cout << "\nEnter a value (double) for rho: ";
162 //  cin >> rho;
163
164     N << 50 << 100 << 150 << 200 << 250 << 300 << 350 << 400 << 450 << 500;
165
166     myfile.open(filename);
167     myfile << setiosflags(ios::showpoint | ios::uppercase);
168     myfile << "rho: " << rho << endl;
169     myfile << "Tolerance: " << epsilon << endl;
170     myfile << right << setw(4) << setfill(' ') << " n: ";
171     myfile << right << setw(30) << setfill(' ') << "  Eigenvalues: ";
172     myfile << right << setw(16) << setfill(' ') << "  Arma Time: ";
173     myfile << right << setw(16) << setfill(' ') << "  Jacobi Time: ";
174     myfile << right << setw(12) << setfill(' ') << "  Transforms: " << endl;
175
176     for (int j = 0; j < 10; j++){
177
178         n = N(j);
179         loops = 0;
180         h = rho / n;
181         e = -1.0/(h*h);
182
183         vec d(n-1);
184         vec p(n+1);
185         vec eigenvalues(n-1);
186         mat A(n-1,n-1);
187         A.zeros();
188         mat M(n-1,n-1);
189         M.eye();
```

```
190
191         for (int i = 0; i <= n; i++){
192
193             p(i) = i*h;
194
195         }
196
197         for (int i = 0; i < n-1; i++){
198
199             d(i) = 2/(h*h) + p(i+1)*p(i+1);
200
201         }
202
203         for (int i = 0; i < n-1; i++){
204
205             A(i,i) = d(i);
206
207             if (i != n-2){
208
209                 A(i,i+1) = e;
210                 A(i+1,i) = e;
211
212             }
213
214         }
215
216         mat B = A;
217
218         clock_t start_Jacobi , end_Jacobi;
219         int k;
220         int l;
221         double MaxOffDiag = MaxOffDiagonal(A, &k, &l, n-1);
222
223         start_Jacobi = clock();
224
225         while (MaxOffDiag > epsilon && loops < maxloops){
226
227             MaxOffDiag = MaxOffDiagonal(A, &k, &l, n-1);
228             Rotate(A, M, k, l, n-1);
229             loops ++;
230
231         }
232
233         end_Jacobi = clock();
234         jacobi_time = (end_Jacobi - start_Jacobi)
235                       /(double)CLOCKS_PER_SEC;
236
237         for (int i = 0; i < n-1; i++){
238
239             eigenvalues(i) = A(i,i);
240
241         }
242
243         eigenvalues = sort(eigenvalues);
```

```
244
245         clock_t start_Armadillo , end_Armadillo;
246         start_Armadillo = clock();
247         mat eigenvector;
248         vec eigenvalue;
249         eig_sym(eigenvalue , eigenvector , B);
250         end_Armadillo = clock();
251         armadillo_time = (end_Armadillo - start_Armadillo)
252                          /(double)CLOCKS_PER_SEC;
253
254         myfile << right << setw(4) << setfill(' ') << n;
255         myfile << right << setw(6) << setfill(' ') << "(" << eigenvalues(0)
256                << ", ";
257         myfile << right << setw(6) << setfill(' ') << eigenvalues(1) << ", ";
258         myfile << right << setw(6) << setfill(' ') << eigenvalues(2) << ")";
259         myfile << right << setw(16) << setfill(' ') << armadillo_time << " s";
260         myfile << right << setw(16) << setfill(' ') << jacobi_time << " s";
261         myfile << right << setw(12) << setfill(' ') << loops << endl;
262
263     }
264
265     myfile.close();
266     return 0;
267
268 }
```

../Code/RhoDependency.py

```
1  '''
2  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3          Project 2 - Schrodinger's Equation for two electrons in
4                      a three-dimensional harmonic oscillator well
5
6          Part b)   - Comparing the dependency on rho and n to have
7                      four leading digits for the eigenvalues.
8                      This means for the lowest three eigenvalues,
9                      the computation returns at least
10                     (2.999,6.999,10.99)
11 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
12 '''
13
14 import math
15 import numpy as np
16 from numpy import linalg as LA
17 import matplotlib.pyplot as plt
18 from matplotlib import colors
19
20 rho = np.array([5.0,6.0,7.0,8.0,9.0,10.0])
21 n = np.array([200.0,250.0,300.0,350.0,400.0,450.0])
22 m , b = np.polyfit(rho,n,1)
23
24 fig , ax = plt.subplots(1)
25
26 hfont = {'fontname':'Times New Roman','size':'14'}
```

```
27
28 plt.plot(rho,m*rho+b,color='black',label="$u_1$",linewidth=1.0)
29 plt.plot(rho,n,ls='None',color='orange',marker='8',markeredgewidth=0.0)
30
31 plt.xlim(4.75,10.25)
32 plt.ylim(175,475)
33
34 ax.set_xlabel('$\\rho_{{max}}$',**hfont)
35 ax.set_ylabel("Square Matrix Dimension $n$",**hfont)
36
37 plt.savefig('RhoDepend.png')
38 plt.show()
```

../Code/2eWavefunctions.cpp

```
1  /*
2  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3          Project 2 - Schrodinger's Equation for two electrons in
4                      a three-dimensional harmonic oscillator well
5
6          Part d)   - Solving the 2-electron wavefunction of coordinate
7                      rho with varying omega_rho
8
9          Results are saved to a text file.
10 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
11 */
12
13 #include <iostream>
14 #include <fstream>
15 #include <iomanip>
16
17 #include <string>
18 #include <time.h>
19
20 #include <cmath>
21 #include <armadillo>
22
23 using namespace std;
24 using namespace arma;
25
26 ofstream myfile;
27
28 int main(){
29
30     int n;
31
32     cout << "Dimensionality of matrix (n): ";
33     cin >> n;
34
35     double omega;
36
37     cout << "Value for Omega_r: ";
38     cin >> omega;
39
```

```
40      string filename;
41
42      cout << "Enter a name for the file: ";
43      cin >> filename;
44
45      double rho = 5.0;
46      double h;
47      double el;
48
49      h = rho / n;
50      el = -1.0/(h*h);
51
52      mat A(n-1,n-1);
53      vec p(n+1);
54      vec z(n+1);
55
56      for (int i = 0; i < n; i++){
57
58          p(i) = i*h;
59
60      }
61
62      for (int i = 0; i < n-1; i++){
63
64          z(i) = pow(omega,2)*pow(p(i+1),2) + 1.0/p(i+1) + 2.0/(pow(h,2));
65
66      }
67
68      for (int i = 0; i < n-1; i++){
69
70          A(i,i) = z(i);
71
72          if (i != n-2){
73
74              A(i,i+1) = el;
75              A(i+1,i) = el;
76
77          }
78      }
79
80      vec eigenvalues;
81      mat eigenvectors;
82
83      eig_sym(eigenvalues,eigenvectors,A);
84
85      myfile.open(filename);
86      myfile << setiosflags(ios::showpoint | ios::uppercase);
87      myfile << right << setw(10) << setfill(' ') << "rho" << endl;
88
89      for (int i = 0; i < n-1; i++){
90
91          myfile << setw(15) << setfill(' ') << setprecision(10) << p(i+1);
92          myfile << setw(15) << setfill(' ') << setprecision(10)
93                 << eigenvectors.col(0);
```

```
        myfile << setw(15) << setfill(' ') << setprecision(10)
               << eigenvectors.col(1);
        myfile << setw(15) << setfill(' ') << setprecision(10)
               << eigenvectors.col(2) << endl;

    }

    myfile.close();
    return 0;

}
```

../Code/2eGraphs.py

```
'''
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        Project 2 - Schrodinger's Equation for two electrons in
                    a three-dimensional harmonic oscillator well

        Part d)   - Solving the 2-electron wavefunction of coordinate
                    rho with varying omega_rho

                    This program reads text files with normalized
                    eigenvectors as a function of rho and graphs
                    the normalized dimensionless wavefunctions.

                    Adjust the name of the file and graph as needed

                    Wavefunctions are graphed and saved to png.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
'''

import math
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
from matplotlib import colors

def getWavefunction(filename):

    file = open(filename, 'r')
    useful = file.readlines()

    rho = []
    u1 = []
    u2 = []
    u3 = []

    for line in useful:

        p, v1, v2, v3 = line.split()
        rho.append(float(p))
        u1.append(float(v1))
        u2.append(float(v2))
```

```
        u3.append(float(v3))

    rho = np.array(rho)
    u1 = np.array(u1)
    u2 = np.array(u2)
    u3 = np.array(u3)

    return rho, u1, u2, u3

x, y1, y2, y3 = getWavefunction('NormalWavefuncOmega0p01.txt')

fig , ax = plt.subplots(1)

hfont = {'fontname':'Times New Roman','size':'14'}

plt.plot(x,y1,color='indianred',label="$u_0$",linewidth=2.0)
plt.plot(x,y2,color='darkviolet',label="$u_1$",linewidth=2.0)
plt.plot(x,y3,color='forestgreen',label="$u_2$",linewidth=2.0)

ax.set_xlabel('Radial Coordinate $\\rho$',**hfont)
ax.set_ylabel("Wavefunction $|{u(\\rho)}|^2$",**hfont)
ax.legend(loc='upper right',fancybox='True',prop={'size':12})

plt.savefig('WavefunctionOmega0p01.png')
plt.show()
```

References

[1] M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2015).

[2] M. Taut, Phys. Rev. A 48, 3561 - 3566 (1993).