Project 2 - FYS3150

Vidar Skogvoll

September 25, 2014

Contents

1 Introduction

Quantum theory is often thougth of as "the most precisely tested and most successful theory in the history of science" [1], which is not a controversial statement. But the despite the success of the theory, the mathematical foundation upon which it is built is more or less the same as it was 50 years ago. However, the computational power available has grown exponatially the last 30 years, allowing us to explore the mysteries of quantum systems within a couple of minutes of computational time. The motivation for this project is to explore one such quantum system by approximating it to a linear algebra eigenvalue problem and look at the properties of such an approximation.

In this project I have written a class in c++-language in which I can enter the properties of the discretization of a system with a particle in a spherical symmetric potential and and find the energy eigenstates of the system. The concrete systems to which this class has been applied are described in the theory section (sections 2.1) I have programmed the Jacobi algorithm (described in section 2.2) to solve the eigenvalue problem of the class.

The systems are discretized and so we might expect some deviation from what is the exact solutions of the systems. That is why I have also investigated the precision of the methods as functions of the discretization limits. Another aspect of the algorithm used is the efficacy of it, and how much time it uses to perform a specific task when compared to other methods. This has also been investigated.

When solving the exercise I noticed some strange properties of the discretization which I have done some preliminary investigation on as a final remark and possible continnuation project. One example of such a property is that the eigenvalue solver seems to return the very first (as in lowest energy) eigenstates. Why is this? If we include a potential in which we have degenerate energies, will this phenomenon still occur?

2 Theory

2.1 The physical quantum system

The bound energy states of the hamiltonian are composed of two parts. The spherical harmincal functions and what we will describe as the *radial part* $R(r)$. It can be shown, using separation of variables that the radial part must satisfy the equation below

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r) \tag{2.1.1}$$

Now introducing new variables $u(r)=rR(r)$ and $\rho=r/\alpha$ it is possible to rewrite the equation above as

$$-\frac{d^2}{d\rho^2}u(\rho)+V(\rho)u(\rho)=\lambda u(\rho) \tag{2.1.2}$$

For some clever choice of the constants α and λ dependent on the potential where V is the effective potential defined as

[1] http://www.4physics.com/phy_demo/QM_Article/article.html

$$V(\rho) = V_0(\rho) + \frac{l(l+1)}{\rho^2} \tag{2.1.3}$$

and $V_0(\rho)$ is the real potential of ρ. It is equation 2.1.3 this project is trying to solve for different potentials.

2.1.1 Discretization and linear algebra

We may discretize, for $i = 0, 1, 2, ..., n_{step}$, $u \to u_i$, $\rho \to \rho_i$ and $V(\rho) \to V_i$ and use the standard expression for the second derivative of a function u.

$$u'' = \frac{u(\rho+h) - 2u(\rho) + u(\rho-h)}{h^2} + O(h^2), \tag{2.1.4}$$

where h is our step length. Introducing the boundary conditions that $u(0) = 0$ and $u(\rho_{max}) = 0$ lets us transform (as shown in project 1) equation 2.1.3 into the linear algebra problem given by equation 2.1.5.

$$\begin{pmatrix} \frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{n_{\text{step}}-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{n_{\text{step}}-1} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\text{step}}-1} \end{pmatrix}$$
$$= \lambda \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\text{step}}-1} \end{pmatrix} \tag{2.1.5}$$

Which is nothing but an eigenvalue problem which we will use Jacobi's method to solve (see section 2.2).

2.1.2 One particle in a harmonic oscillator

For a particle in a three dimensional harmonic oscillator, the potential is given as

$$V_0(r) = \frac{1}{2} m\omega^2 r^2 \tag{2.1.6}$$

The smart choice of α and λ (as discussed in 2.1in this case is

$$\alpha = \left(\frac{\hbar}{m\omega}\right)^{1/2} \tag{2.1.7}$$

And

$$\lambda = \frac{2m\alpha^2}{\hbar^2} \tag{2.1.8}$$

Because with these definition, the effective potential becomes

$$V(\rho) = \rho^2 + \frac{l(l+1)}{\rho^2} \tag{2.1.9}$$

This effective potential will from here on be known as the "plain_harmonic" potential. It can be shown theoretically that this potential has eigenvalues λ equal to

$$\lambda = 3 + 3n + 2l \tag{2.1.10}$$

For $n = 0, 1, 2, ...$ and $l = 0, 1, ..., n-1$. We will use this exact result as a measure of the error in the discretization of the problem.

2.1.3 Two electrons in a harmonic oscillator

To simplify the problem we will now ignore the angular momentum quantum number l. The general equation for two electrons with center of mass position $\vec{R}$ given by

$$\vec{R} = \frac{\vec{r}_1 + \vec{r}_2}{2} \tag{2.1.11}$$

And relative position $\vec{r}$ (i.e. position of the one with regards to the other) given by

$$\vec{r} = \vec{r}_1 - \vec{r}_2 \tag{2.1.12}$$

Where $\vec{r}_1$ and $\vec{r}_2$ are the positions operators for each of the electrons can be shown to be (see section A in the appendix)

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} - \frac{\hbar^2}{4m}\frac{d^2}{dR^2} + \frac{1}{4}kr^2 + kR^2\right) u(r,R) = E^{(2)}u(r,R). \tag{2.1.13}$$

Now, using separation of variables, and adding the repulsive potential between the electrons , it is possible to show that the relative distance wave function $\psi(\rho)$ of the two electrons must satisfy the following equation

$$-\frac{d^2}{d\rho^2}\psi(\rho) + V(\rho)\psi(\rho) = \lambda\psi(\rho) \tag{2.1.14}$$

Where $\rho = r/\alpha$ as before and $V(\rho)$ is the effective potential defined as

$$V(\rho) = \omega_r^2\rho^2 + \frac{1}{\rho} \tag{2.1.15}$$

Where the three terms on the left hand side stems from the oscillator potential and the repulsive coulomb potential respectively. The smart choices of α, λ and ω_r to obtain the equation above are as follows

$$\alpha = \frac{\hbar^2}{m\beta e^2} \tag{2.1.16}$$

$$\lambda = \frac{m\alpha^2}{\hbar^2}E \tag{2.1.17}$$

$$\omega_r^2 = \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4 \tag{2.1.18}$$

Where $\beta e^2 = 1.44eV\ nm$. ω_r is a constant characterizing the strength of the harmonic oscillator. As we see, this is just a modification of the one dimensional problem discussed earlier with a different potential, and we can use the eigenvalue solving mechanism to solve it. The potential

$$\omega_r^2\rho^2 + \frac{1}{\rho} \tag{2.1.15}$$

Will from here on be known as the "two_elec" potential.

2.2 The jacobi rotation algorithm

Suppose we have a linear algebra eigenvalue problem

$$A\vec{x} = \lambda\vec{x} \tag{2.2.1}$$

and some simularity transformation such that

$$S^T AS = D \tag{2.2.2}$$

Where S is a unitary matrix and D is a diagonal matrix. Linear algebra tells us that if A is real and symmetric, there is always such a matrix and the Jacobi rotation algorithm finds us one. The method is explained at lengths in the lecture notes of the course [1], but for this section's purpose, it suffices to say that the algorithm provides a set of unitary matrices $S_0, S_1, ..., S_N$ so that

$$A_N = S^T AS = S_N^T ... S_1^T A_0 S_1 ... S_N = D \tag{2.2.3}$$

At each iteration we save the matrix A_i we get from calculation $A_i = S_i^T A_{i-1} S$. Now, let's look at the eigenvalue problem with variable notation familiar with quantum mechanics.

$$H\psi = E\psi \tag{2.2.4}$$

Here, H is a real, symmetric matrix, ψ is a eigenvector and E is a eigenvalue. Applying S^T from the left hand side

$$S^T H\psi = ES^T\psi \tag{2.2.5}$$

Since S is a unitary matrix we get

$$S^T H S S^T \psi = E S^T \psi \tag{2.2.6}$$
$$D S^T \psi = E S^T \psi \tag{2.2.7}$$

As we see, the eigenvalues E of the diagonal matrix are the same as the eigenvalues of the initial matrix A. But since D is diagonal we can just read the eigenvalues off of it. The eigenvectors of the diagonal matrix are given by the identity matrix, but are also given as we see from equation 2.2.7 as $S^T\psi$ (if ψ is a matrix with eigenvector colums) , thus

$$S^T \psi = I \tag{2.2.8}$$
$$S S^T \psi = SI \tag{2.2.9}$$
$$\psi = S \tag{2.2.10}$$

At each iteration we may save the matrix $V_i = S_i^T V_{i-1}$ where $V_0 = I$ and get

$$V = S_N^T S_{N-1}^T ... S_1^T \tag{2.2.11}$$

If we transpose this matrix we get

$$V^T = S_1 S_2 ... S_N = \psi \tag{2.2.12}$$

And then we know how to extract the eigenvectors, as well as the eigenvalues, from the algorithm. Since unitary matrices are norm preserving [2] and I is (obviously) an orthonormal matrix, the matrix V^T will also be orthonormal which ensures that the states we get from this method are normalized.

The S matrices are known as jacobi rotation matrices and each similarity transformation is known as a rotation. Within the algorithm for the jacobi method there is a tolerance ϵ involved which lets us know when to stop the algortihm from running, i.e. when we think our diagonal matrix is "diagonal enough". The algorithm stops when the largest off-diagonal element m squared is bigger than the given tolerance, in other words, when $m^2 > eps$. Througout this project, $\epsilon = 10^{-10}$.

3 Method

3.1 The error in the eigenvalues as a function of n_{step} and ρ_{max}

In reality, the coordinate ρ can take any value between 0 and ∞. This is, for obvious reasons, not a possibility when discretizing and using nummerical methods. Therefore we must explicitly set the value of ρ_{max} at which point the wave function should be zero.

This is an approximation. So is the number n_{step} of discretization steps we take, and so this part will deal with the error in the eigenvalues of the "plain_harmonic" potential (given by equation 2.1.10) as a function of n_{step} and ρ_{max}.

The script **error_nstep_rhomax.cpp** (see section B.2) investigated the error in the obtained eigenvalues as a function of different n_{steps} and ρ_{max}. l was set to equal 0 for simplicity.

[2] Source: http://en.wikipedia.org/wiki/Unitary_matrix

3.2 Number of rotations as a function of dimensionality

The number of rotations needed in the Jacobi algorithm to meet the required tolerance is expected to increase as a function of the dimensionality, i.e. the discretization resolution.

The script **number_rotations_dim.cpp** (see section B.3) investigated number of rotations needed as a function of dimensionality n_{step} with a "plain_harmonic"-potential with $l = 0$.

3.3 Timely difference between jacobi algorithm and another algorithm

Another interesting aspect of this problem is to compare the execution time of the jacobi algorithm with another algorithm.

The script **compare_times_arma_jacobi.cpp** (see section B.4) investigated the time used to solve the problem with $\rho_{max} = 6$ and the "plain_harmonic"-potential with $l = 0$ for different values of n_{step}. And compared the elapsed time from the jacobi algorithm to that of the algorithm named "eig_sym" provided by the armadillo library.

3.4 Degeneration with $l \neq 0$

In a real system with a three dimensional harmonic oscillator, there is, for $l \neq 0$ a degeneracy of the energies. This because there are two quantum numbers, l and m, involved in the spherical harmonical functions and for each l m can take values $m = -l, -l+1, ..., l-1, l$, while the energy, as we may read from equation 2.1.10, is not dependent on m. An interesting question is whether we will see this degeneration when computing the eigenenergies of our system or not.

The script **degeneration.cpp** (see section B.5) investigated this issue.

3.5 The two-electron system

We will now look at some of the specifics of the two-electron system with the repulsive potential as described in section 2.1.3.

The script **two_electrons.cpp** (see section B.6) solved problem of the relative distance between the two electrons for three different values of ω_r (three different harmonical potential strengths).

4 Results and discussion

4.1 The error in the eigenvalues as a function of n_{step} and ρ_{max}

Figure 4.1.1 shows the results from running the script described in section 3.1.

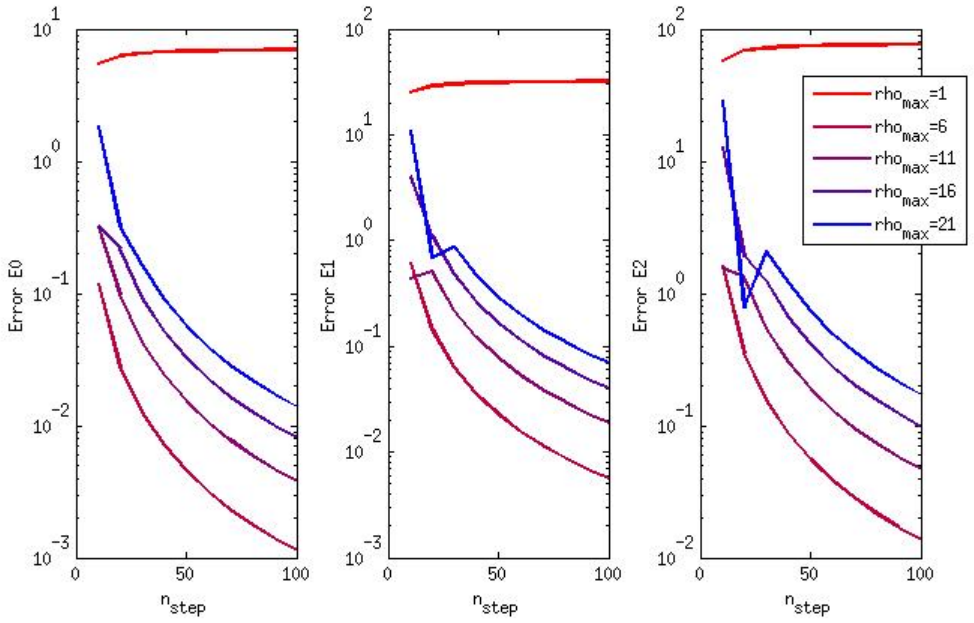

Figure 4.1.1: Semilogaritmic plot of the absolute error in the estimation of the first three eigenvalues $E_0 = 3$, $E_0 = 7$ and $E_0 = 7$. For $\rho_{max} \geq 6$ the precision increases as a function of the resolution n_{step} but decreases as a function of ρ_{max}.

The figure shows several insteresting aspects. Firstly, it clearly demonstrates that for a too small value of ρ_{max}, in this case $\rho_{max} = 1$ the number n_{steps} of discretization steps does not matter. The error is present and doesn't change considerably with increasing n_{step} When the value of ρ_{max} gets larger however, we see that the precision of the estimation gets better with increased number n_{step} of discretization points, which is what one would expect.

Secondly, we see that the precision in the approximatiom of these first few eigenvalues decrease as ρ_{max} increases. This is not surprising, seeing as the first few eigenfunctions (to which these eigenvalues apply) decrease in magnitude when ρ becomes larger. Having more points in the area where the eigenfunctions really matter, i.e. near 0 - small ρ_{max}, would naturally cause better estimations of the eigenvalues as well.

Finally, we see that the error in general increases with which increasing eigenvalues. This is because the eigenfunctions become more complicated for the more excited states and more points are needed to represent them with the same precision.

4.2 Number of rotations as a function of dimensionality

Figure 4.2.1 shows the results from counting the number of rotations as a function of the matrix dimensionality n_{step}.

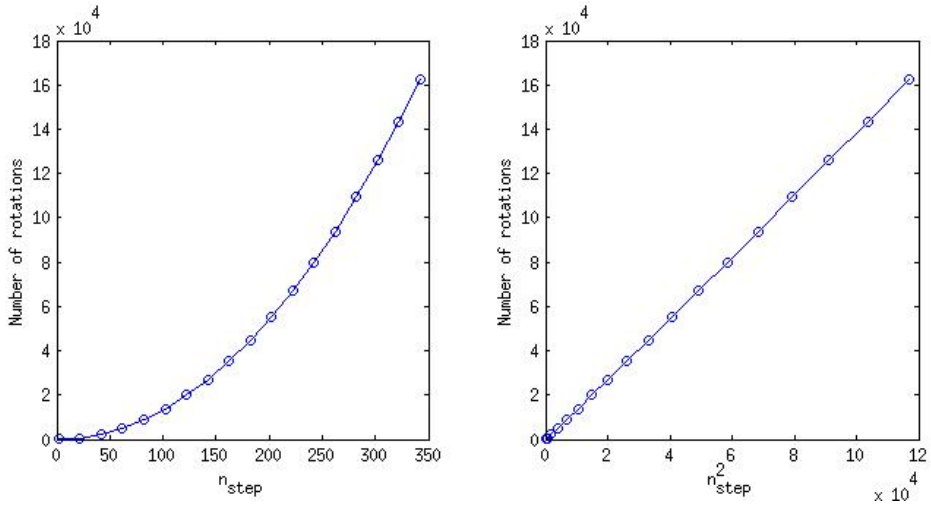

Figure 4.2.1: Plot of the number of rotations (see section 2.2) used in the jacobi algorithm against the dimensionality n_{step} of the matrix. The right hand plot shows a plot of the number of rotations as a function of dimensionality n_{step} of the matrix squared. The number of rotations increase with the dimensionality of matrix, and there seems to be a quadratic trend.

As expected, the number of rotations increases as a function of n_{step}. From the right hand side of the figure, it seems like a fair assumption that the number of rotations $Rot(n_{step})$ needed to obtained the wanted accuracy is proportional to the dimensionality n_{step} squared, i.e.

$$Rot(n_{step}) = kn_{step}^2 \tag{4.2.1}$$

Where k is some proportionality constant. An estimate of this constant based upon the data obtained is given in table 4.2.1.

What	Value
k	1.376 ± 0.006

Table 4.2.1: The estimation of the proportionality constant between jacobi rotations and the dimensionality n_{step} of the matrix squared (equation 4.2.1). The error estimation is made using MATLAB's curve fitting tool with a confidence interval of 95%

The narrow confidence interval in the estimation of the proportionality constant k confirms that the number of rotations rot is indeed a quadratic function of the dimensionality n_{step} of the matrix. This is also in agreement with the lecture notes which states that "one needs typicically $3n^2 - 5n^2$ rotations [...] to zero out the non-diagonal elements" [3]. The reason for this problem

[3] Lecture notes [1], p.217

needing only $1.376n^2$ rotation may be due to most off-diagonal elements being 0 in the first place or that our tolerance for the largest off-diagonal element is high.

An interesting question for another project would be to investigate how this proportionality holds for other tolerances in the jacobi rotation algorithm. And if so, how the proportionality constant k depends on the tolerance of the algorithm.

4.3 Timely difference between jacobi algorithm and another algorithm

The times needed to solve the problem with the different algorithms are given in table 4.3.1.

Dimensionality n_{step}	Jacobi algorithm time [s]	Armadillo algorithm time [s]
10	9.4×10^{-5}	3.0×10^{-5}
30	6.0×10^{-3}	2.4×10^{-4}
50	4.1×10^{-2}	7.5×10^{-4}
70	1.2×10^{-1}	1.5×10^{-3}
90	3.6×10^{-1}	2.6×10^{-3}
110	7.8×10^{-1}	4.1×10^{-3}
130	1.4×10^{0}	6.2×10^{-3}
150	2.7×10^{0}	8.8×10^{-3}
170	4.5×10^{0}	1.2×10^{-2}
190	7.0×10^{0}	1.6×10^{-2}
210	1.0×10^{1}	2.1×10^{-2}

Table 4.3.1: The times needed to solve the eigenvalue problem using the jacobi- and the armadillo algorithm as a function of the matrix dimensionality n_{step}. Time elapsed increases with n_{step} and it seems that the algorithm used by armadillo is much more efficient than the jacobi algorithm.

A plot of the elapsed times from table 4.3.1 is given in figure 4.3.1.

As we see from the results, the algorithm provided by the armadillo lbirary is much more efficient when it comes to time elapsed than the jacobi algorithm. The time elapsed is many orders of magnitudes larger for the jacobi algorithm than the armadillo algorithm.

As we see, the time elapsed for the jacobi algorithm is apperantly proportional to the dimensionality n_{step} of the matrix to the fourth power. As we saw in the section above, the number of rotations needed for the algorithm to finish went as n^2_{step}. This indicates, given that there is a proportionality between flops and time elapsd, that the number of floating points operations for each rotation goes as n^2_{step}. This, however, is *not* in agreement with the lecture notes which states that "each rotation requires $4n$ operations" [4]. An interesting topic for another time would be to investigate where this discrepency between this theoretical number of flops and time elapsed arises.

The figure (4.3.1) also show that the time elapsed for the algorithm provided by armadillo goes pretty much as $(n_{step})^{2.5}$.

[4] Lecture notes [1], p.217

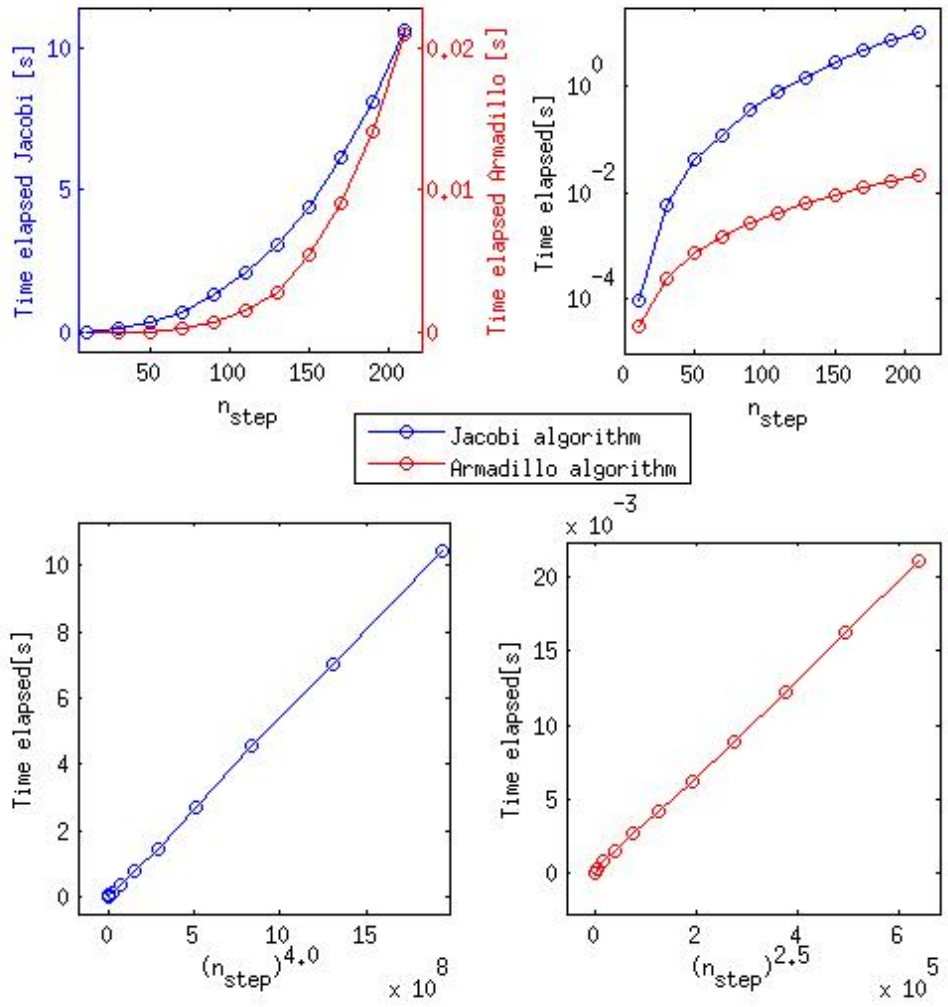

Figure 4.3.1: Plot of the elapsed times from solving the eigenvalue problem with different algorithms (see table 4.3.1) with different axes in different plots. We see that the algorithm used by the armadillo library is much more efficient than the jacobi algorithm and that the they depend differently with respect to the dimensionality n_{step}.

4.4 Degeneration with $l \neq 0$

Running the script and looking at the eigenvalues revealed that the degeneration was not caught by this method of solving the problem. This is not surprising; The quantum number m does not affect the radial part of the states, so the radial parts for the different states corresponding to different values of m are the same. Since we are solving an eigenvalue problem with a real symmetric matrix we are guarantied to find n_{step} linearly independent eigenvectors (i.e. radial functions) and thus we simply couldn't have found the degeneracy.

4.5 The two-electron system

A plot of the results from running the script is shown in figure 4.5.1.

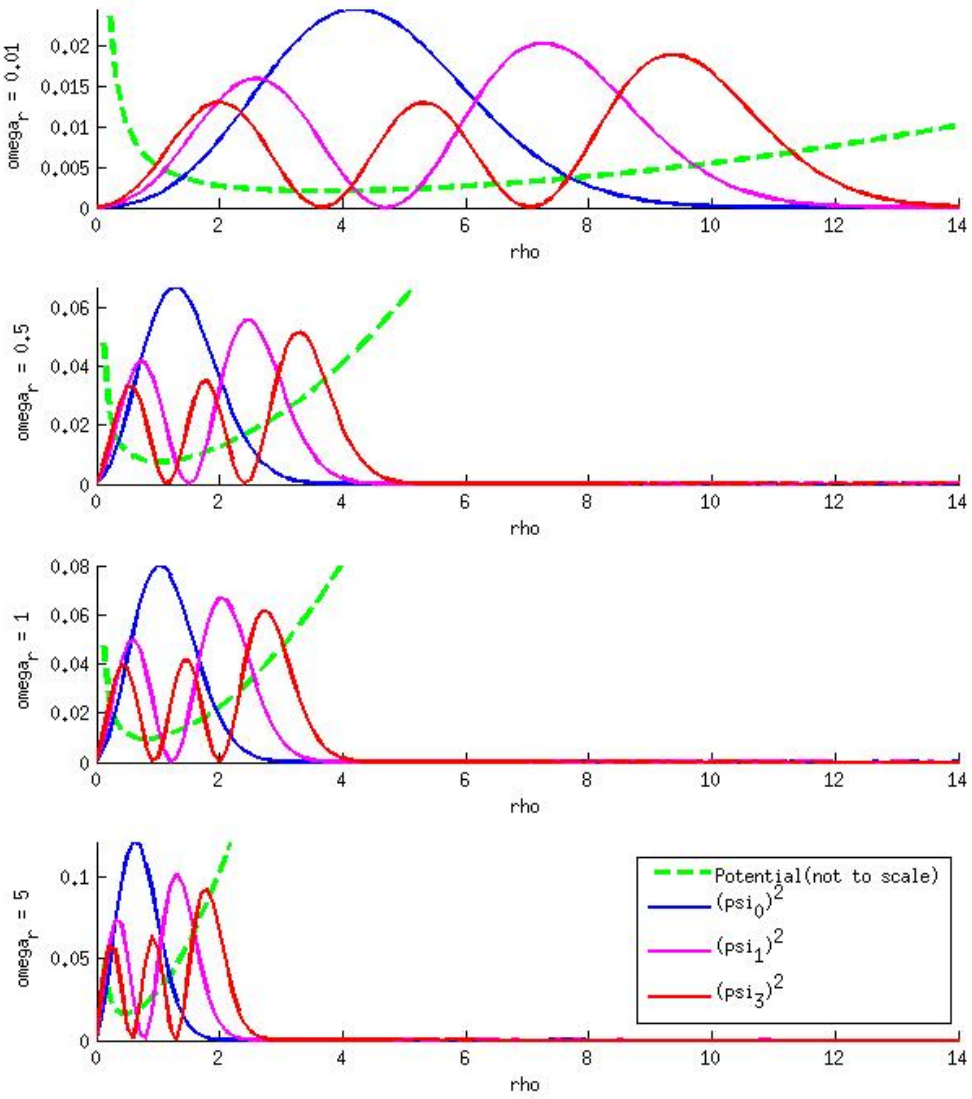

Figure 4.5.1: Plot of the probability density ψ_i^2 as a function of the distance between the two electrons for the three first eigenstates (ψ_0, ψ_1 and ψ_2) and four different harmonic oscillator potential strengths (given by $\omega_r = 0.01, 0.5, 1, 5$). We see that the electrons tend to be far apart for weaker harmonic oscillator potentials and closer for the lowest energy states.

As we see from the figure the probability density of the distance between the electrons gets pressed towards 0 as the potential "well" gets tighter, i.e. when the harmonic oscillator strengt ω_r gets bigger. This makes sense because a strong harmonic oscillator potential will draw the electrons very close together untill the repulsive coloumb force between them will equal the pull of the harmonic oscillator.

We also see that the higher energy-states allows the electrons to be further apart. This is also natural since much energy is needed to be far apart in a potential "well".

5 Conclusion

Appendix

A Deducing the equation for the two electron system.

The following reasoning has been shamelessly copy-pasted from the exercise instructions made by Morten Hjorth-Jensen, professor at UiO.

We will now study two electrons in a harmonic oscillator well which also interact via a repulsive Coulomb interaction. Let us start with the single-electron equation written as

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r) + \frac{1}{2}kr^2u(r) = E^{(1)}u(r),$$

where $E^{(1)}$ stands for the energy with one electron only. For two electrons with no repulsive Coulomb interaction, we have the following Schrödinger equation

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

Note that we deal with a two-electron wave function $u(r_1, r_2)$ and two-electron energy $E^{(2)}$.

With no interaction this can be written out as the product of two single-electron wave functions, that is we have a solution on closed form.

We introduce the relative coordinate $\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2$ and the center-of-mass coordinate $\mathbf{R} = 1/2(\mathbf{r}_1 + \mathbf{r}_2)$. With these new coordinates, the radial Schrödinger equation reads

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

B Codes

All the codes used in this project can also be found at https://github.com/vidarsko/Project2.

B.1 The library file

Header file: project2lib.h

```
#ifndef project2lib_h
#define project2lib_h
#include <iostream>
#include <armadillo>
#include <cmath>
#include <ctime>
#include <fstream>
#include <string>
using namespace std;
using namespace arma;

//Function for finding eigenvalues and eigenvectors using the jacobi method
void eig_jacobi(vec& eigval,mat& eigvec,mat A,int N, int& rotation_counter, double tol = 1e-10);
//Help functions for completing that task
vec odmmi(mat A,int N); // Off Diagonal Max Matrix Indices

//Class for storing and solving the quantum problem
class SphericalQuantum{
    /*
    Class which stores the basic properties of a quantum system with a spherical
    symmetric potential. It inclues functions which finds the energy levels of the
    bound states and the corresponding eigenstates.
    */
    private:
        int n_step, rotation_counter;
        double rho_min,rho_max, h;
        vec rho,V,Energies;
        mat Energy_states,H;
    public:
        //Constructor
        SphericalQuantum (int a, double b, double c=0);

        //**************************Member functions************************//
        //Configure functions
        void set_potential(vec potential);

        //Solvefunction
        void solve(string method="jacobi");

        //Data extraction functions
        vec get_rho(void);
        vec get_V(void);
        mat get_H(void);
        vec get_lambda(int number=3);
        int get_rotation_counter(void);

        //Test and print functions
        void testprint(void);
        void print2file(void);
};

//Different potential functions
vec plain_harmonic(vec x,int l = 0);
vec two_elec(vec x,double omega_r=1);
#endif
```

Class file: project2lib.cpp

```
#include "project2lib.h"

void eig_jacobi(vec& eigval,mat& eigvec,mat A,int N,int& rotation_counter,double tol)
    {
    /*
    Function that takes a (nxn) matrix A and returns its eigenvalues
    and corresponding eigenvectors as column vectors
    stored in eigval and eigvec respectively.
    Input:
    - vec& eigval -                     n-vector for storing eigenvalues
    - mat& eigvec -                     (nxn).matrix for storing eigenvectors as
        colums.
    - mat A -                           the (nxn)-matrix we're interested in.
    - int N -                           the dimension of the square A matrix.
    - int& rotation_counter -           integer for storing the rotation_counter
    - tol [optional] -                  the tolerance for the off-diagonal elements.
        (default=1e-10)
    */

    //Allocation of memory
    mat B = A,R(N,N);
    vec klm;
    double m,tau,t,c,s,s2,c2,sc,r_il,r_ik;
    int k,l;

    //Initial error and R matrix,
    R.eye();
    klm = odmmi(A,N);
    m = klm(2);
    rotation_counter=0;
    //While loop until tolerance criteria is met.
    while (m>tol){
        //Counter
        rotation_counter++;

        //Print error to estimate time developement
        //cout << m << endl;

        //Finding the largest off-diagonal element indices
        k = klm(0),l = klm(1);

        //Jacobi rotate values
        tau = (A(l,l) - A(k,k))/(2*A(k,l));
        if (tau>=0){
            t = 1/(tau + sqrt(1 + tau*tau));
        }
        else {
            t = -1/(-tau+sqrt(1+tau*tau));
        }
        c = 1/sqrt(1+t*t);
        s = t*c;

        //Rotation product B
        s2 = s*s; c2 = c*c; sc = s*c;
        B(k,k) = A(k,k)*c2 - 2*A(k,l)*sc + A(l,l)*s2;
        B(l,l) = A(l,l)*c2 + 2*A(k,l)*sc + A(k,k)*s2;
        B(k,l) = 0;
        B(l,k) = 0;

        for (int i = 0;i<N;i++){
            if (i!=k && i!=l){
                B(i,k) = A(i,k)*c - A(i,l)*s;
                B(k,i) = B(i,k);
                B(i,l) = A(i,l)*c + A(i,k)*s;
                B(l,i) = B(i,l);
            }
            //Update eigenvectors
            r_ik = R(i,k);
            r_il = R(i,l);
            R(i,k) = c*r_ik - s*r_il;
```

```
68              R(i,l) = c*r_il + s*r_ik;
69          }
70
71          //Update A and error estimate
72          A = B;
73          klm = odmmi(A,N);
74          m = klm(2);
75      }
76
77      //Fill inn eigenvalues and eigenvectors
78      for (int i = 0; i<N; i++){
79          eigval(i) = A(i,i);
80      }
81      eigvec = R;
82
83      //Sort
84      uvec sort_indices = sort_index(eigval);
85      mat eigvec_copy(N,N);    eigvec_copy = eigvec;
86      vec eigval_copy(N);      eigval_copy = eigval;
87      for (int i = 0;i<N;i++){
88          eigvec.col(i) = eigvec_copy.col(sort_indices(i));
89          eigval(i) = eigval_copy(sort_indices(i));
90      }
91  }
92
93  vec odmmi(mat A,int N){
94      /*
95      Off Diagonal Max Matrix Indices:
96      Function that takes a square matrix A and returns the indices as a
97      of the maximum off-diagonal element of the A matrix and its value m.
98      Input:
99      - mat A - The matrix in question
100     - int N - The dimension of the square matrix
101     Output:
102     - vec klm - A 3-vector (k,l,m) consisting of the indices of the maximum squared
103     element m and m itself.
104     */
105     vec klm = zeros(3);
106     double m = 0;
107     for (int i = 0; i<N; i++){
108         for (int j = 0; j<N; j++){
109             if (A(i,j)*A(i,j)>m && i != j){
110                 m = A(i,j)*A(i,j);
111                 klm(0) = i;
112                 klm(1) = j;
113                 klm(2) = m;
114             }
115         }
116     }
117     return klm;
118 }
119
120
121 //*****************SphericalQuantum class functions**********************//
122 //Constructor
123 SphericalQuantum::SphericalQuantum(int a, double b, double c){
124     /*
125     Constructor for the Spherical quantum class.
126     Takes arguments
127     int a - n_step, the number of steps. (i.e. the resolution)
128     double b - rho_max, the largest value of rho.
129     double c - rho_min [default=0], the smallest value of rho.
130     */
131     n_step = a; rho_max=b; rho_min=c;
132     h = (rho_max-rho_min)/n_step;
133     rho = linspace(rho_min,rho_max,n_step+1); //position array
134 }
135
136 //Configuration functions
137 void SphericalQuantum::set_potential(vec potential){
138     /*
139     Function that adds the potential to the system.
140     */
```

```
141     V = potential;
142 }
143
144 //Solve functions
145 void SphericalQuantum::solve(string method){
146     /*
147     Solves the system using specified method.
148     Input:
149     - string method:
150         "jacobi" (default)  -uses the jacobi rotation method.
151         "armadillo"         -uses the method provided by the armadillo library.
152     */
153     //Dimensions on hamiltonian and energystates
154     H = zeros(n_step,n_step);
155     Energies = zeros(n_step);
156     Energy_states = zeros(n_step,n_step);
157
158     double ode = -1/(h*h); //off-diagonal element
159     double h2 = h*h;
160     //Set Hamiltonian matrix elements
161     for (int i = 0;i<n_step;i++){
162         H(i,i) = 2/h2 + V(i+1); //V goes from 0 to n_step, we want the middle part.
163         if      (i==0)          {H(i,i+1) = ode;}
164         else if (i==n_step-1)   {                H(i,i-1) = ode;}
165         else                    {H(i,i+1) = ode; H(i,i-1) = ode;}
166     }
167     //Find the eigenvalues and states
168     if (method == "jacobi"){
169         eig_jacobi(Energies,Energy_states,H,n_step,rotation_counter);
170     }
171     else if (method == "armadillo"){
172         eig_sym(Energies,Energy_states,H);
173     }
174 }
175
176 //Data extraction functions
177 vec SphericalQuantum::get_rho(void){
178     /*
179     Function that returns the position vector of the system.
180     */
181     return rho;
182 }
183
184 vec SphericalQuantum::get_V(void){
185     /*
186     Fucntion that returns the potential of the system.
187     */
188     return V;
189 }
190
191 mat SphericalQuantum::get_H(void){
192     /*
193     Function that returns the hamiltonian of the system.
194     */
195     return H;
196 }
197
198 vec SphericalQuantum::get_lambda(int number){
199     /*
200     Function that returns the (number) first eigenvalues of the system as a vec.
201     Input:
202     - int number [default=3]: number of eigenvalues to be returned.
203     */
204     vec E_return (3);
205     for (int i=0;i<number;i++){
206         E_return(i) = Energies(i);
207     }
208     return E_return;
209 }
210
211 int SphericalQuantum::get_rotation_counter(void){
212     /*
```

```
213     Function that returns the number of rotations performed during the jacobi ↩
            algorithm.
214     */
215     return rotation_counter;
216 }
217
218
219 //Test and print functions
220 void SphericalQuantum::testprint(void){
221     /*
222     Prints out values of different variables.
223     */
224     cout << "rho_min = " << rho_min << endl;
225     cout << "rho_max = " << rho_max << endl;
226     cout << "n_step = " << n_step << endl;
227     cout << "h =" << h << endl;
228 }
229
230 void SphericalQuantum::print2file(void){
231     /*
232     Prints the solved system to a datafile in the following format:
233     Filename: "SphericalQuantum-[Date and time].out"
234     Content:
235     """
236     rho      ,V      ,psi_0   , ...    ,psi_{n_{step}} ,E
237     0,       ,0.33   ,0.00    , ...    ,0.00           ,E_0
238     h,       ,0.423  ,0.21    , ...    ,0.11           ,E_1
239     ...
240     rho_max,0.22     ,0.00    , ...    ,0.00           ,E_{n_step}
241     """
242     */
243     //Syntax to get time stamp
244     time_t timer = time(NULL);
245     struct tm * timeinfo  = localtime(&timer);
246     char TID[80];
247     strftime(TID,80,"%c",timeinfo);
248
249     //Open file syntax
250     ofstream myfile;
251     string tmp = "SphericalQuantum-"; tmp.append(string(TID));
252     tmp.append(".out"); const char* name = tmp.c_str();
253     myfile.open(name);
254
255     //Print data to file
256     //Top row
257     myfile << "rho, V, ";
258     for (int i=0;i<n_step;i++){myfile<<"psi_"<<i<<", ";}
259     myfile << "E\n";
260
261
262     //Content
263     //Zero position
264     myfile << rho(0) << ", " << V(0) << ", ";
265     for (int i=0;i<n_step;i++){myfile << "0, ";}
266     myfile << Energies(0)<<",\n";
267     for (int i = 1; i<n_step; i++){ //the middle part
268         myfile << rho(i) << ", " << V(i) << ", "; //We want the middle part
269         for (int j=0;j<n_step;j++){myfile << Energy_states(i,j) << ", ";}
270         myfile << Energies(i)<< ",\n";
271     }
272
273     //Close file
274     myfile.close();
275 }
276
277
278 //******************Potential functions *****************//
279 vec plain_harmonic(vec x, int l){
280     /*
281     The plain harmonic potential.
282     Input:
283     - vec x:            position coordinates
284     - int l(default=0): angular momentum quantum number
```

```
285     */
286     return x%x + l*(l+1)/(x%x);
287 }
288
289 vec two_elec(vec x,double omega_r){
290     /*
291     The potential used in the two-electron problem.
292     Input:
293     - vec x:                position coordinates
294     - omega_r (default=1):  strength of the harmonic oscillator
295     */
296     return omega_r*x%x + 1/x;
297
298 }
```

B.2 The error in the eigenvalues as a function of n_{step} and ρ_{max}

error_nstep_rhomax.cpp

```cpp
#include <iostream>
#include <armadillo>
#include <fstream>
#include "project2lib.h"
using namespace std;
using namespace arma;

int main(){
    ofstream output;
    output.open("error_nstep_rhomax.out");

    int     n_step_min = 10,    n_step_max = 100,   n_step_indent=10;
    double  rho_max_min = 1,    rho_max_max = 25,   rho_max_indent=5;
    int number_eig_max = 3;

    //Title_bar
    output << "n_step, ";
    for (double rho_max=rho_max_min;rho_max<rho_max_max;rho_max+=rho_max_indent){
        for (int number_eig = 0; number_eig<number_eig_max;number_eig++){
            output << "rho_max" << rho_max << "_E" << number_eig << ", ";
        }
    }
    output << '\n';

    //Content
    for (int n_step=n_step_min; n_step <=n_step_max; n_step +=n_step_indent){
        output << n_step << ", ";
        cout << n_step << endl;
        for (double rho_max=rho_max_min;rho_max<rho_max_max;rho_max+=rho_max_indent){
            SphericalQuantum system(n_step, rho_max);
            vec rho = system.get_rho();
            system.set_potential(plain_harmonic(rho));
            system.solve();
            vec E = system.get_lambda(number_eig_max);
            for (int i=0;i<number_eig_max;i++){
                output << E(i) << ", ";
            }
        }
        output << '\n';
    }
    output.close();
    return 0;
}
```

B.3 Number of rotations as a function of dimensionality

number_rotations_dim.cpp

```cpp
#include <iostream>
#include <armadillo>
#include <fstream>
#include "project2lib.h"
using namespace std;
using namespace arma;

int main(){

    //Output results
    ofstream output;
    output.open("number_rotations_dim.out");
    output << "n_step ," << "Rotations \n";

    //Parameteres
    double rho_max = 10;

    for (int n_step = 2; n_step<360; n_step +=20){
        cout << n_step << endl;
        SphericalQuantum system(n_step,rho_max);
        vec rho = system.get_rho();
        system.set_potential(plain_harmonic(rho));
        system.solve();
        output << n_step << ", " << system.get_rotation_counter() << '\n';
    }
    output.close();
}
```

B.4 Timely difference between the jacobi algorithm and another algorithm

compare_times_arma_jacobi.cpp

```
#include <iostream>
#include <armadillo>
#include "project2lib.h"
#include <time.h>
#include <fstream>
using namespace std;
using namespace arma;

int main(){
    ofstream output;
    output.open("compare_times_arma_jacobi.out");
    output << "n_step, jacobi, armadillo" << '\n';
    int rho_max = 6;
    for (int n_step = 10; n_step <= 210; n_step += 20){
        cout << n_step<< endl;
        //The need for numerous iterations decrease as n_step gets bigger.
        int k_iterations = 10000/n_step;

        SphericalQuantum system(n_step,rho_max);
        vec rho = system.get_rho();
        system.set_potential(plain_harmonic(rho));

        //Jacobi
        clock_t start_jacobi = clock(); //Time stamp
        for (int k=0; k<k_iterations;k++){
            system.solve("jacobi");
        }
        clock_t end_jacobi = clock();
        long double time_jacobi = (end_jacobi-start_jacobi)/(k_iterations * (double)↩
            CLOCKS_PER_SEC);


        //Armadillo
        clock_t start_armadillo = clock();
        for (int k=0; k<k_iterations;k++){
            system.solve("armadillo");
        }
        clock_t  end_armadillo = clock();
        long double time_armadillo = (end_armadillo-start_armadillo)/(k_iterations * ↩
            (double)CLOCKS_PER_SEC);

        output << n_step << ", " << time_jacobi << ", " << time_armadillo << '\n';

    }
    output.close();
}
```

B.5 Degeneration with $l \neq 0$

degeneration.cpp

```
#include <iostream>
#include <armadillo>
#include "project2lib.h"
using namespace std;
using namespace arma;

int main(){
    int n_step = 100;
    double rho_max = 10;
    SphericalQuantum system(n_step,rho_max);
    vec rho = system.get_rho();
    system.set_potential(plain_harmonic(rho,1));
    system.solve();
    system.print2file();
}
```

B.6 The two-electron system

two_electrons.cpp

```
#include <iostream>
#include <armadillo>
#include "project2lib.h"
using namespace std;
using namespace arma;

int main(){
    //Different omega_r's to be tested
    vec omega_rs(4);
    omega_rs(0) = 0.01; omega_rs(1) = 0.5; omega_rs(2)=1.0; omega_rs(3)=5;

    //Rho_max's for the different omega_r's
    double rho_max = 20;

    //Dimension
    int n_step = 200;

    for (int i=0;i<4;i++){
        double omega_r = omega_rs(i);
        SphericalQuantum system(n_step,rho_max);
        vec rho = system.get_rho();
        system.set_potential(two_elec(rho,omega_r));
        system.solve();
        system.print2file();

    }
}
```

References

[1] Morten Hjorth-Jensen. *Computational Physics - Lecture Notes Fall 2014*. August 2014.