FYS4150 - Computational Physics
Project 2:
Schrödinger's equation for two electrons in a three dimensional harmonic oscillator well

Henrik Schou Røising

October 2, 2015

Abstract

In this project we study the numerical (radial) solution the three dimensional Schrödinger equation, first for a single electron and then for two interacting electrons in a harmonic oscillator potential. We first implement Jacobi's rotation algorithm and then move over to Armadillo's standardized solver for eigenvalue problems due to computation speed. For the analytical solvable problem, the numerical found wave functions and eigenvalues are compared to exact solutions and found to be in agreement to at least four leading digits for a number of grid points equal to $n = 350$. For the two-electron case results are compared to [2]. Also here the results are found to be in good agreement. We find that the relative wave functions spread out as the relative oscillator frequency is decreased, just as one would expect classically.

- Github repository link with all source files: https://github.com/henrisro/Project2

1 Introduction

We start out with the dimensionless Schrödinger equation for an electron in a harmonic oscillator potential. By approximating the second derivative with a three point formula, the problem is shown to be equivalent to an eigenvalue problem for a tridiagonal matrix. This is implemented by the use of Jacobi's rotation algorithm first, and then by the use of standardized algorithms in the Armadillo library for some gain of computation speed. The three lowest energy eigenvalues and eigenvectors are found and compared to exact results. Then the problem is extended to involve two interacting electrons in a harmonic oscillator potential. The lowest lying energy states are also here plotted and compared to some special analytical results from [2].

2 Theory

2.1 Single electron

The radial Schrödinger equation for a single electron reads:

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

We will take $V(r) = m\omega^2 r^2/2$ to be the harmonic oscillator potential with corresponding energies in three dimensions: $E_{nl} = \hbar\omega(2n + l + 3/2)\ \forall n, l \in \mathbb{N}_0$. We will consider only zero angular momentum states, $l = 0$, from now on. By introducing a series of variables, this can be brought to a simple dimensionless form:

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

where $u(r) = R(r)/r$ and $\rho = r/\alpha$ is the dimensionless length parameter. The constant α is fixed to $\alpha = (\hbar/m\omega)^{1/2}$ and the energy parameter is $\lambda = 2m\alpha^2E/\hbar^2$. By introducing this, it is quickly seen that $\lambda \in \{3, 7, 11, \dots\}$. In the following we will look closer at the ground state with $\lambda_0 = 3$ and the two first excited states, $\lambda_1 = 7$ and $\lambda_2 = 11$. We further approximate the second derivative in equation (2),

$$\frac{d^2u(\rho)}{d\rho^2}=\frac{u(\rho+h)-2u(\rho)+u(\rho-h)}{h^2}+\mathcal{O}(h^2), \tag{3}$$

with the step length, h, defined as $h = (\rho_{\max} - \rho_{\min})/n$ for some discretization integer n. Let also $u_i = u(\rho_{\min} + ih)$, so that equation (2) can be written, by the use of (3), as

$$-\frac{1}{h^2}u_{i-1}+\left(\frac{2}{h^2}+\rho_i^2\right)u_i-\frac{1}{h^2}u_{i+1}=\lambda u_i \tag{4}$$

$$A\boldsymbol{u}=\lambda\boldsymbol{u} \tag{5}$$

for $i \in \{2, \dots, n-2\}$, where we defined $\boldsymbol{u} = (u_1, u_2, \dots, u_{n-1})^T$ and

$$A=\begin{bmatrix} 2/h^2+\rho_1^2 & -1/h^2 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -1/h^2 & 2/h^2+\rho_2^2 & -1/h^2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1/h^2 & 2/h^2+\rho_3^2 & -1/h^2 & \cdots & 0 & 0 & 0 \\ & \vdots & & & \ddots & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & -1/h^2 & 2/h^2+\rho_{n-2}^2 & -1/h^2 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1/h^2 & 2/h^2+\rho_{n-1}^2 \end{bmatrix} \tag{6}$$

So the problem is in this nomenclature to find the eigenvectors (i.e. wave functions) corresponding to the eigenvalues λ_0, λ_1 and λ_2 for the matrix A. Before moving on to the two interacting electrons, we will state the exact wave functions for the the three lowest eigenvalues for later reference and comparison. The general form of these radial wave functions is:

$$R_{n,l=0}(r)=N_n e^{-\frac{m\omega}{2\hbar}r^2}L_n^{1/2}\left(-\frac{m\omega}{2\hbar}r^2\right), \tag{7}$$

where $L_n^{1/2}$ refers to an associated Laguerre polynomial[1]. By evaluating this for the three lowest energy states and normalizing them according to $\int_0^\infty dr\, |rR_{n,l=0}(r)|^2 = \int_0^\infty d\rho\, |u_n(\rho)|^2 = 1$, one finds the following:

$$|u_0(\rho)|^2 = \frac{4}{\sqrt{\pi}}\rho^2 e^{-\rho^2} \tag{8}$$

and

$$|u_1(\rho)|^2 = \frac{8}{3\sqrt{\pi}}\rho^2 \left(\frac{3}{2} - \rho^2\right)^2 e^{-\rho^2} \tag{9}$$

and

$$|u_2(\rho)|^2 = \frac{8}{15\sqrt{\pi}}\rho^2 \left(\frac{15}{4} - 5\rho^2 + \rho^4\right)^2 e^{-\rho^2} \tag{10}$$

2.2 Two electrons

Now assume that two non-interacting electrons are subject to the same harmonic potential as before (still with zero angular momentum). The radial Schrödinger equation (1) is now modified to

$$\left(-\frac{\hbar^2}{2m}\left(\frac{d^2}{dr_1^2} + \frac{d^2}{dr_2^2}\right) + \frac{1}{2}m\omega^2(r_1^2 + r_2^2)\right) u(r_1, r_2) = E^{(2)}u(r_1, r_2) \tag{11}$$

Similar to before one can reduce this equation by introducing the relative radial coordinate, $\boldsymbol{r} = \boldsymbol{r}_1 - \boldsymbol{r}_2$, the center of mass coordinate, $\boldsymbol{R} = (\boldsymbol{r}_1 + \boldsymbol{r}_2)/2$, and assuming separation of variables $u(r, R) = \psi(r)\phi(R)$ ($E^{(2)} = E_r + E_R$ for the energy). This leads to the following equation for the relative wave function, $\psi(r)$, when we also include the repulsive Coulomb interaction, $V(r) = \beta e^2/|\boldsymbol{r}_1 - \boldsymbol{r}_2| = \beta e^2/r$ ($\beta = 1.44$ eVnm):

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dr^2} + \frac{1}{2}m\omega^2 r^2 + \frac{\beta e^2}{r}\right)\psi(r) = E_r\psi(r) \tag{12}$$

One can make this dimensionless by introducing the same variable as before, $\rho = r/\alpha$, but this time fix $\alpha = \hbar^2/(m\beta e^2)$. Introduce also a relative frequency, $\omega_r^2 = \left(\frac{m\omega}{2\hbar}\alpha^2\right)^2$ and an energy parameter, $\lambda = \frac{m\alpha^2}{\hbar^2}E_r$. This results in a form analogous to (2):

$$-\frac{d^2\psi(\rho)}{d\rho^2} + \left(\omega_r^2\rho^2 + \frac{1}{\rho}\right)\psi(\rho) = \lambda\psi(\rho) \tag{13}$$

We thus see that by discretization, the only difference to the matrix A in (6) are the diagonal elements which now take the form $d_i = \frac{2}{h^2} + \omega_r^2\rho_i^2 + \frac{1}{\rho_i}$.

[1]https://en.wikipedia.org/wiki/Particle_in_a_spherically_symmetric_potential, downloaded 28.09.15.

3 Method / Algorithm

In this section we focus on Jacobi's rotation algorithm which is to be applied in order to diagonalize matrix A in equation (6). However: the standardized diagonalization routine in the Armadillo library, `eig_sym()`, will be used to produce wave functions in practice due to the slowness of Jacobi's method when the dimension, n, becomes of the order of 10^2.

3.1 Jacobi's rotation algorithm

Given the eigenvalue problem $A\boldsymbol{x} = \lambda\boldsymbol{x}$. The idea of Jacobi's rotation algorithm [1] is to apply a series of similarity transformations to both sides of this equation (preserving the eigenvalues and the initially chosen orthogonality of eigenvectors). More specifically we will apply S with $S^T = S^{-1}$ as an Euclidean rotation matrix around one of the basic axes until the non-diagonal elements of A have been transformed away, leaving a matrix with the eigenvalues on the diagonal. Say that the axis of rotation is orthogonal to the plane spanned by $\boldsymbol{e}_k$ and $\boldsymbol{e}_l$, where $\{\boldsymbol{e}_i\}$ denotes the Euclidean set of basis vectors. We can then take the non-zero elements of S to be parametrized by an angle θ: $S_{kk} = S_{ll} = \cos\theta$, $S_{kl} = -S_{lk} = -\sin\theta$ and $S_{ii} = 1$ for $i \neq k$ and $i \neq l$. Then the iterative process $A_{j+1} = S^T A_j S$ will be applied until the *Frobenius norm* of $A_{j+1} - A_j$ becomes smaller than some tolerance ϵ. The Frobenius norm is defined as

$$\|A\|_F = \sqrt{\sum_{i,j} |A_{ij}|^2} \tag{14}$$

Define now $\cot(2\theta) = \tau = (A_{ll} - A_{kk})/(2A_{kl})$. By demanding that the off-diagonal elements A_{kl} after a similarity transformation should be zero, one finds two possible values of $\tan\theta$: $t = \tan\theta = -\tau \pm \sqrt{1+\tau^2}$. In order to choose the optimal value of t, observe that

$$\left\| S^T A S - A \right\|_F^2 = 4(1-c) \sum_{i,i\neq k,l}^{n} (A_{ik}^2 - A_{il}^2) + \frac{2A_{kl}^2}{c^2} \tag{15}$$

with $c = \cos\theta = 1/\sqrt{1+t^2}$ (and $s = \sin\theta = tc$). This means that the value of c closest to 1 is the optimal choice, which again means choosing t to be the smaller (absolute) value of the two possible. This can be compactly written:

$$t = \frac{\mathrm{sgn}(\tau)}{|\tau| + \sqrt{1+\tau^2}} \tag{16}$$

Observe further that when choosing this $|t| \leq 1$, we have that $c \leq 1/\sqrt{2}$ such that $|\theta| \leq \pi/4$ by the relations above. When all this is to be applied to a C++ program, we will structure the main function as follows:

Listing 1: Code sample: main function with use of Jacobi's rotation algorithm. The computation time is also calculated.

```
// Calculate time used by Jacobi method:
clock_t start_Jacobi, finish_Jacobi;
// Indices determining axis of rotation:
```

```
int k, l;
double diag_maximum = off_diag_abs(A, &k, &l, n-1);
start_Jacobi = clock();
// Continue until off-diagonal elements are smaller than tolerance eps or
    iteration limit is reached:
while (diag_maximum > eps && counter < max_counter) {
    diag_maximum = off_diag_abs(A, &k, &l, n-1);
    Jacobi_rotate(A, R, k, l, n-1);
    counter++;
}
finish_Jacobi = clock();
calculation_time_Jacobi = (finish_Jacobi - start_Jacobi)/(double)CLOCKS_PER_SEC;
```

Here `off_diag_abs(mat A, int * k, int * l, int n)` is a function that determines the indices k and l at which $|A_{kl}|$ is maximal (this gives the optimal choice for the next similarity transformation [1]). Further the function `Jacobi_rotate(mat & A, mat & R, int k, int l, int n)` applies one similarity transformation to the matrix $S^T AS$ with the choice of τ, t, c and s discussed above. To avoid large matrix multiplications (in Armadillo library), the changed elements are coded by hand with the following function core:

Listing 2: Code sample: function core of `Jacobi_rotate()`. The matrix R contains the eigenvectors as they are updated with the same similarity transformation as A. Initially R is chosen to be the identity matrix. This implementation is brute force when considering a tridiagonal matrix and is close to the version found in the lecture notes (2015) [1].

```
// Declaration of local variables to save computation time:
double R_ik, R_il, A_ik, A_il;
double A_kk = A(k,k);
double A_ll = A(l,l);

// Apply rotation by hand:
A(k,k) = A_kk*c*c - 2*A(k,l)*c*s + A_ll*s*s;
A(l,l) = A_ll*c*c + 2*A(k,l)*c*s + A_kk*s*s;
// Elements transformed to zero:
A(k,l) = 0.0;
A(l,k) = 0.0;
for (int i = 0; i < n; i++) {
   if (i != k && i != l) {
         A_ik = A(i,k);
         A_il = A(i,l);

                  A(i,k) = A_ik*c - A_il*s;
                  A(k,i) = A(i,k);
                  A(i,l) = A_il*c + A_ik*s;
                  A(l,i) = A(i,l);
   }
   // Update eigenvectors:
   R_ik = R(i,k);
   R_il = R(i,l);
   R(i,k) = R_ik*c - R_il*s;
   R(i,l) = R_il*c + R_ik*s;
}
```

3.2 Lanczos' algorithm

It would also be appropriate to mention another algorithm that was implemented during the work with this project. Lanczos' algorithm [1] is an iterative process that transforms a matrix A, by the use of a similarity transformation $T = Q^T AQ$, to a tridiagonal matrix T. If one let $Q = [\boldsymbol{q}_1, \dots, \boldsymbol{q}_n]$ be written as a composition of its (orthogonal) column vectors, and denote $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_n)^T$ as the diagonal part of T and $\boldsymbol{\beta} = (\beta_1, \dots, \beta_{n-1})^T$ the next-to-diagonal part of T, the criterion $T = Q^T AQ \Rightarrow QT = AQ$ can be written:

$$A\boldsymbol{q}_k = \beta_{k-1}\boldsymbol{q}_{k-1} + \alpha_k \boldsymbol{q}_k + \beta_k \boldsymbol{q}_{k+1} \tag{17}$$

and $\beta_0 \boldsymbol{q}_0 = 0$ for $k \in \{1, n-1\}$. Orthogonality of the vectors, $\boldsymbol{q}_i^T \boldsymbol{q}_j = \delta_{ij}$, implies that

$$\alpha_k = \boldsymbol{q}_k^T A \boldsymbol{q}_k \tag{18}$$

Thus the two equations (17) and (18) together gives us the basic recipe for determining the vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ (and hence the tridiagonal matrix T). The algorithm is similar to that of Householder, and basically becomes the same if one chooses the vector $\boldsymbol{q}_1 = \boldsymbol{e}_1$ (generally it could be a random, normed vector). It can be shown that the extremal eigenvalues of T becomes iteratively better estimates of the eigenvalues of A. A code sample of the basic algorithm (implemented in a a very direct way) is shown in listing below, and a brief discussion of the method can be found in the sections for results and discussion. It should be clear at this point that this method applied to our problem is somewhat of an overkill since the matrix we are considering (6) is already tridiagonal. In addition Lanczos' algorithm should be supplemented with an effective method of calculating eigenvalues of tridiagonal matrices, and that has not been, nor will be, the primary focus of this project.

Listing 3: Code sample: The main body of the function `Lanczos(mat & A, int n)` from the file `Project2_f_Lanczos.cpp` found in the github repository. It was written based on the lecture notes [1].

```
// tolerance in beta values:
double btol = 1E-5;
// initiate vectors and matrices used in the loop:
int kmax = n;
int k = 0;
vec alpha = zeros(kmax);
vec beta = zeros(kmax);
vec q_k = zeros(n);
vec q1 = zeros(n);
//q1(0) = 1;
// Generate random normed vector:
q1 = randu<vec>(n);
q1 /= norm_by_hand(q1,n);
vec r = q1;
double b = 1;

// Loop over k:
while (b > btol && k < kmax) {
    k += 1;
    vec qkm1 = q_k;
    q_k = r/b;
    vec Aqk = A*q_k;
    // Update alpha element:
    alpha(k-1) = dot(trans(q_k),Aqk);
    r = Aqk - q_k*alpha(k-1) - qkm1*b;
    b = norm_by_hand(r,n);
    // Update beta element:
    beta(k-1) = b;
}
```

4 Results

4.1 Single electron

The Jacobi method was implemented according to the discussion in section 3.1. The implementation was tested against the standard method `eig_sym()` in the Armadillo library to ensure that it was working properly. The exact eigenvalues of the single electron problem are as mentioned known, so we can use the three lowest of these, $\{3, 7, 11\}$, for calibration of the number of grid points n and the end point of the interval $\rho_{\max}$ needed to get solid results. By trial and error, the end point $\rho_{\max}$ was set to 5 as the three lowest lying wave functions will turn out to be well localized within this value. The tolerance of the Jacobi method was set to $\epsilon = 10^{-8}$, which for practical purposes is small enough.

A table of results, both showing the resulting eigenvalues and time consumption, is shown in table 1. From the table one can see that as n passes about 200, all three eigenvalues are correct to four leading digits. Furthermore the last column shows the number of similarity transformations used in the Jacobi method. A rough estimate based on the table values shows that the method uses about $N(n) \approx 1.6n^2$ similarity transformations to diagonalize the matrix A ($(n-1) \times (n-1)$ dimension of matrix). A more sophisticated interpolation can of course be done for the values in table 1, but we are here only interested in an approximate behaviour. The number $N(n)$ is typically of this order, with a slightly larger constant of proportionality when the matrix is not tridiagonal [1]. The wave functions corresponding to the eigenvalues shown in table 1 could be plotted for validation of the results. This is what we have done in figure 1. We can see a satisfactory agreement already for $n = 350$.

4.2 Two interacting electrons

The program written for the single electron problem was adjusted to handle two interacting electrons according to equation (13) by a slight change in the potential on the diagonal on matrix A. The program could then be tested against the (partially) exact results shown in Table I in [2]. In table 2 we have shown the computed ground state energy along with the approximate formula ε'_m from equation (16a) in [2] (which is valid only for small oscillator frequencies, ω_r). For the two special values $\omega_r \in \{1/20, 1/4\}$, exacts results are provided by the formula $\varepsilon'_{\text{int}}$ in the same article.

Grid points, n	Three lowest eigenvalues calculated with Jacobi's rotation method	Calculation time, T_{arma} [s]	Calculation time, T_{Jacobi} [s]	Number of similarity transformations
50	$\{2.99687, 6.98434, 10.9619\}$	0.00588	0.0146110	3852
100	$\{2.99922, 6.99609, 10.9907\}$	0.003089	0.307990	16217
150	$\{2.99965, 6.99827, 10.9960\}$	0.006838	1.65875	36847
200	$\{2.99980, 6.99903, 10.9978\}$	0.0126010	5.54378	66065
250	$\{2.99988, 6.99938, 10.9987\}$	0.0183590	13.2021	103791
300	$\{2.99991, 6.99957, 10.9991\}$	0.0310610	28.2012	149879
350	$\{2.99994, 6.99968, 10.9994\}$	0.0430640	51.8338	204649

Table 1: Table showing three lowest eigenvalues calculated with Jacobi rotation method as discussed in the method section. In calculating this we set $\rho_{\text{max}} = 5$ (it will turn out that this is large enough for the three lowest states). Tolerance in Jacobi method was set to $\epsilon = 10^{-8}$. This table was produced by running the program `Project2_a.py` which compiles and runs `Project2_a.cpp`.

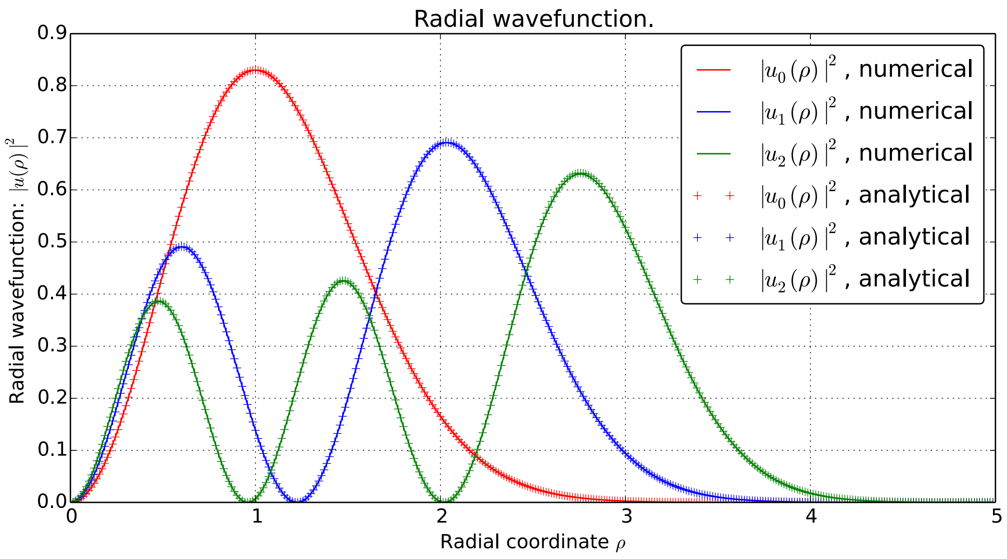

Figure 1: Plot of radial wave functions corresponding to calculated eigenvalues in table 1 by the use of Armadillo's solver, `eig_sym()`, with $n = 350$ and $\rho_{\text{max}} = 5$. The numerical results are plotted together with the analytical results from section 2.1, given in equations (8), (9) and (10). This figure was produced by running `Project2_a_plot_eigenfunctions.py` which compiles and runs `Project2_a_plot_eigenfunctions.cpp`.

The wave functions corresponding to (some of) the eigenvalues seen in table 2 where normalized so that $\int_0^\infty |u(\rho)|^2 = 1$. In figure 2 they are plotted for the choices $\omega_r = 0.01$ and $\omega_r = 0.5$, and correspondingly reasonable choices of ρ_{max}. Similar plots for $\omega_r = 1$ and $\omega_r = 5$ can be seen in figure 3.

Oscillator frequency, ω_r	Numerically found ground state eigenvalue, λ	Reference article [2] approximate formula, $2\varepsilon'$	Reference article [2] improved formula, $2\varepsilon'_{\text{int}}$
0.01	0.1058	0.1050	- -
0.05	0.3500	0.3431	0.3500
0.25	1.2500	1.1830	1.2500
0.5	2.2301	2.0566	- -
1.0	4.0578	3.6219	- -
5.0	17.4484	14.1863	- -

Table 2: Table showing numerically computed ground state eigenvalue with approximate (and some exact) results from the reference article [2] (the factor of 2 in front of the article formulas are due to a slightly different formulation of the problem in the article). In producing these results we constantly chose $n = 350$ while ρ_{max} was varied according to the localization of the ground state. It was chosen to be $\{60, 50, 30, 8, 5, 2\}$ for the different values of ω_r respectively (seen in figure 2 and 3). The values in this table were produced by running the python script `Project2_c_eigenvalues.py` which calls `Project2_c_eigenvalues.cpp` for the different problem parameters.

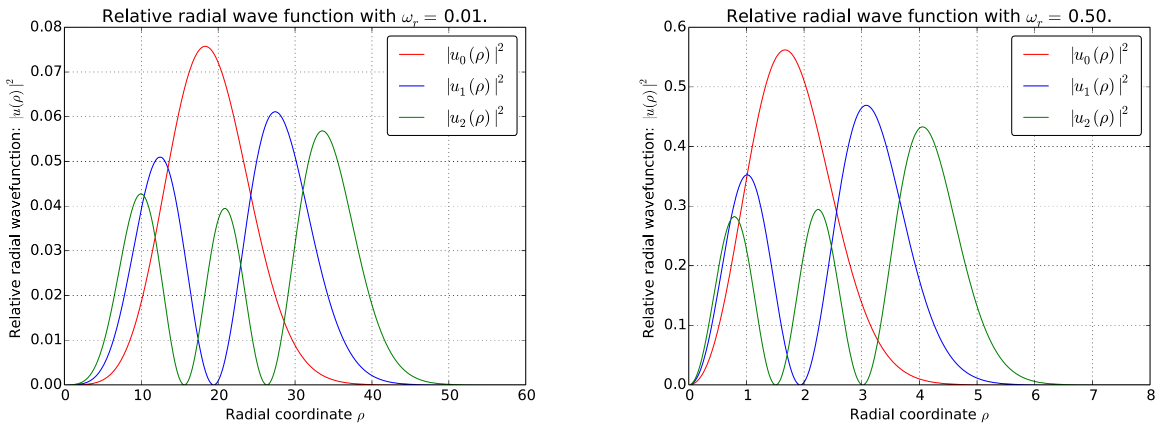

Figure 2: Plots of the three lowest lying wave functions for the values $\omega_r = 0.01$ and $\omega_r = 0.5$. The left figure was produced with $\rho_{\text{max}} = 60$ and the right with $\rho_{\text{max}} = 8$ (both for $n = 350$). The plots were produced by running the python script `Project2_d.py` which compiles and runs `Project2_d.cpp`.

4.3 Lanczos' method

Finally we will mention shortly what was generally seen when Lanczos' algorithm was applied. The corresponding programs (`Project2_f_Lanczos.cpp` and `Project2_f_Lanczos.py`) and benchmark results can be found in the github repository and will not be restated in detail here. Since the original matrix A is tridiagonal, we instead chose to apply Lanczos' method to a random symmetric matrix and compare the eigenvalues of the resulting tridiagonal matrix with those of the matrix itself. The txt-file, `Table_of_results_Lanczos.txt`, contains an example run with a random 50×50 matrix and the eigenvalues of both the matrix itself and the eigenvalues of the tridiagonal matrix returned by the function `Lanczos()`. The last column

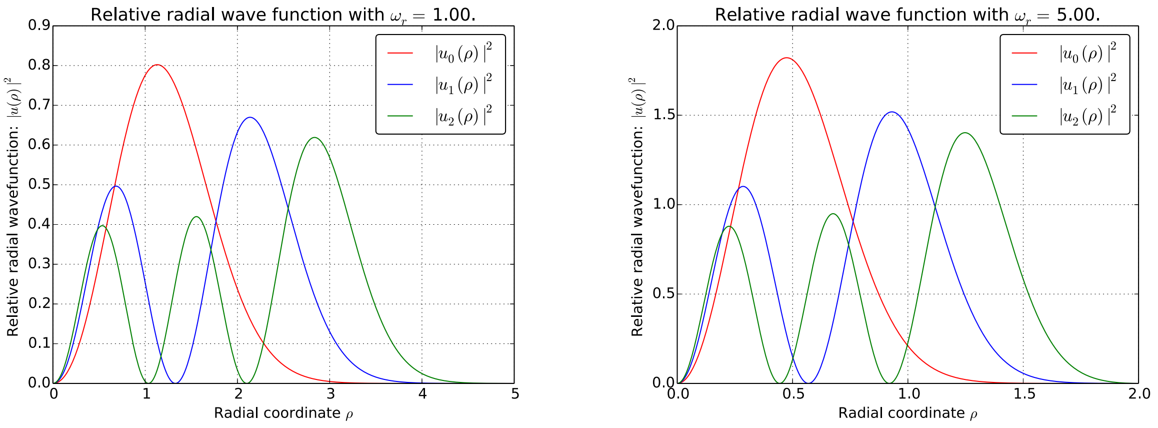

Figure 3: Plots of the three lowest lying wave functions for the values $\omega_r = 1$ and $\omega_r = 5$. The left figure was produced with $\rho_{\max} = 5$ and the right with $\rho_{\max} = 2$ (both for $n = 350$). The plots were produced by running the python script `Project2_d.py` which compiles and runs `Project2_d.cpp`.

of the file shows a clear trend, namely that the extremal eigenvalues are very close to equal, while the eigenvalues towards the middle of the spectrum are found to a very low degree of accuracy. In fact, the largest eigenvalues ends up being degenerate in the tridiagonal matrix - a clear weakness of the method. This problem was seen to increase in the matrix dimension.

5 Discussion

The implementation of Jacobi's rotation algorithm was seen to produce results in good agreement with the analytical eigenvalues, λ_0, λ_1 and λ_2. However we did not consider possible speed ups of the algorithm due to the simple structure of A. For example the search for the largest off-diagonal element, which is a time consuming part of the algorithm, could be optimized since A has identical off-diagonal elements initially. Table 1 reveals anyhow that Jacobi's method used about $1.6n^2$ similarity transformations to make A diagonal, while one typically needs $3n^2$ to $5n^2$ transformations for the full, non-trivial matrix problem [1]. Instead we moved over to Armadillo's standardized solver, `eig_sym()`, to produce and compare eigenfunctions in practice.

A quite large part of the problem concerned how to chose n and $\rho_{\max}$ in order to produce results with reasonably good accuracy. Plots of the wave functions were used as the intuitive measure of how well localized the states were, i.e. where it was appropriate to make the cut-off in ρ. This part of the problem worked as a simple kind of *test* that the problem should have to pass before moving over to the two-interacting case, where the lack of analytical results makes it more demanding to check the validity of the results. Other simple tests were made during the development of the programs as well. For example putting $n = 3$ in `Project2_a.cpp` (corresponding to A being a 2×2 matrix) was seen to result in one needed Jacobi rotation - as expected (to be strict: one should really subtract one from the variable `counter` in the program since the `while` loop has to be passed one extra time after it has reached the final iteration in order for the loop to break). It was also checked that the function `off_diag_abs()` was working properly. This was done by making explicitly one of the off-diagonal matrix elements in A larger in absolute value than the rest. The element's indices were seen to be returned by the function. In addition the orthogonality of the column vectors of the matrix R (initially equal to the identity) were seen to remain orthogonal to floating point precision, even after 100 Jacobi rotations. These tests are not composed in one summarizing test program in the code attachment section, but were instead checked during the development. *Unit tests* (for example provided by `UnitTest++`) were on the other hand not directly used in this project. When considering the two-electron problem, we used the special values of ω_r ($\omega_r = 1/4$ and $\omega_r = 1/20$) in [2] as a testing platform for the modified program `Project2_c_eigenvalues.cpp`. As table 2 shows, the program reproduced the values given by M. Taut to at least four leading digits when the number of grid points was set to $n = 350$.

As the programs passed the simple validity test just mentioned, we can focus on the physics reflected in figure 2 and 3. The parameter ω_r adjusts the strength of the relative oscillator potential. As it was increased, one can clearly see how the lowest three states are squeezed together (the relative radial wave functions), and by turning on/off the Coulomb interaction, we could check that the electrons are on average further apart with repulsive interaction between them. These are both of course classically expected effects. Finally we saw that Lanczos' algorithm was seen to work satisfactory well for extremal eigenvalues (see file `Table_of_results_Lanczos.txt` for a 50×50 example), but rather poorly for eigenvalues in the middle of the spectrum. It would be an overkill to apply this method to A since the matrix already is tridiagonal.

6 Conclusion

In this project we started with an implementation of an analytically solvable system: a single electron in a three dimensional harmonic oscillator potential. The matrix, A, corresponding to the discretized problem was diagonalized with a slow Jacobi rotation method and further with Armadillo's standardized methods. As both the eigenvalues and corresponding eigenfunctions were seen to be in good agreement with analytic results - after trial and error in choosing n and $\rho_{\max}$ - the problem was extended to include a second electron interacting by Coulomb repulsion with the first. This corresponded to a simple change in the diagonal elements in A, so the code could be reused with small modifications.

The calculated wave functions for the two-electron-problem turned out to reveal an intuitively expected behaviour: the relative wave functions are smeared out in the radial direction as the repulsive interaction is included. For some values of the radial oscillator frequency, ω_r, the results could be compared to exact results from [2]. We saw perfect agreement (that is: to four leading digits) for the values of ω_r which had eigenvalues calculated in [2] and took this as a sign of a well functioning program.

References

[1] M. Hjort-Jensen. Computational physics, lecture notes fall 2015. Department of Physics, University of Oslo, 2015.

[2] M. Taut. Two electrons in an external oscillator potential: Particular analytic solutions of a coulomb correlation problem. *Phys. Rev. A*, 48:3561–3566, Nov 1993.

Code attachment

Listing 4: Program: `Project2_a.cpp` which is compiled and ran by calling `Project2_a.py`. It produces the txt-file `Table_of_resutls.txt`.

```
/*
**      Project2: a) and b)
**      A first brute force implementation of Jacobi's rotation algorithm.
**      The method is applied to solving Shr dinger's equation
**      with a harmonic oscillator poteniti al for a single electron.
**      Computation time is compared to Armadillo's standard solver and written
**      to txt-file "Table_of_results.txt".
*/

#include <iostream>
#include <cmath>
#include <armadillo>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <time.h>

using namespace std;
using namespace arma;
ofstream ofile;

// Function to find indices of off-diagonal element in (symmetric) matrix A with
// the largest absolute value. Indices k and l are returned.
double off_diag_abs(mat A, int * k, int * l, int n) {
        double max_val = 0.0;
        double A_ij;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
                A_ij = fabs(A(i,j));
            // Found new element with larger absolute value:
            if (A_ij > max_val) { max_val = A_ij; *k = i; *l = j; }
        }
    }
    return max_val;
}


// Function to make one iteration of Jacobi method.
// Rotation is defined by the indices k and l (returned by the function off_diag_abs).
// Matrix A of dimension n is rotated. Matrix R contains eigenvector basis and is also
    rotated.
// Implementation of this is very close to that found in the lecture notes of FYS4150
    (2015).
void Jacobi_rotate(mat & A, mat & R, int k, int l, int n) {
    double t, tau, s, c;

    if ( A(k,l) != 0.0 ) {
      tau = ( A(l,l) - A(k,k) )/( 2*A(k,l) );

      // Ensure that we get the smallest absolute value of rotation angle:
      if (tau >= 0.0)  { t = 1.0/(fabs(tau) + sqrt(1.0 + tau*tau)); }
      else { t = -1.0/(fabs(tau) + sqrt(1.0 + tau*tau)); }

      c = 1.0/sqrt(1.0 + t*t);
      s = t*c;
    }
    else {
        // If A(k,l) = 0, no need for rotation:
        c = 1.0; s = 0.0;
    }

    // Calculate all rotations by hand to avoid multiplication of large matrices:
    double R_ik, R_il, A_ik, A_il;
    double A_kk = A(k,k);
    double A_ll = A(l,l);

    // Rotation of elements defined by k and l:
    A(k,k) = A_kk*c*c - 2*A(k,l)*c*s + A_ll*s*s;
    A(l,l) = A_ll*c*c + 2*A(k,l)*c*s + A_kk*s*s;
    A(k,l) = 0.0;
    A(l,k) = 0.0;

    // Other elements affected by rotation:
    for (int i = 0; i < n; i++) {
        if (i != k && i != l) {
            A_ik = A(i,k);
            A_il = A(i,l);

                A(i,k) = A_ik*c - A_il*s;
                A(k,i) = A(i,k);
                A(i,l) = A_il*c + A_ik*s;
                A(l,i) = A(i,l);
        }
        R_ik = R(i,k);
        R_il = R(i,l);
        R(i,k) = R_ik*c - R_il*s;
        R(i,l) = R_il*c + R_ik*s;
    }
    return;
}

int main() {

    // Initial variables:
    int n, counter;
    int max_counter = 1E6;
    double eps = 1.0E-8;
    double rho_min, rho_max, h, e;
    double calculation_time_Jacobi, calculation_time_arma;
    string outfilename;
    rowvec N;

    // Consider the following choice of rho_max:
    rho_min = 0.0;
    rho_max = 5.0;

    // Comparison will be made for the following system sizes:
    N  << 50 << 100 << 150 << 200 << 250 << 300 << 350;

    outfilename = "Table_of_results.txt";
    ofile.open(outfilename);
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << "␣␣rho_max␣set␣to:␣" << rho_max << "␣and␣tolerance␣in␣Jacobi␣method␣set␣to
        :␣" << eps << endl;
    ofile << "␣␣␣n:␣␣␣␣␣␣␣␣␣␣Eigenvalues:␣␣␣␣␣␣␣␣␣␣␣T(Armadillo):␣␣␣␣␣T(Jacobi):␣␣␣#
        Transformations:" << endl;
    // Loop over different system sizes:
    for (int j = 0; j < 7; j++) {

    n = N(j);
    counter = 0;

    h = (rho_max - rho_min)/n;
    e = -1.0/(h*h);

    // Initializing matrices and vectors that will be used:
```

```
    vec d(n-1);
    vec rho(n+1);
    vec eigenvalues(n-1);
    mat A(n-1,n-1);
    A.zeros();
    mat R(n-1,n-1);
    // Making R equal to identity initially.
    // Corrsponds to initial choice of basis:
    R.eye();

    for (int i=0; i <= n; i++) { rho(i) = rho_min + i*h; }
    // Now rho(0) = rho_min and rho(n) = rho_max.
    for (int i=0; i < n-1; i++) { d(i) = 2/(h*h) + rho(i+1)*rho(i+1); }
    // Now d(0) is evaluated at rho(1) etc.

    // Initiating matrix A:
    for (int i=0; i < n-1; i++) {
          A(i,i) = d(i);
          if (i != n-2) {
                A(i,i+1) = e;
                A(i+1,i) = e;
          }
    }

    // Make a copy of A that will be used for Armadillo library method:
    mat B = A;

    // Calculate computation time and eigenvectors with Jacobi method:
    clock_t start_Jacobi, finish_Jacobi;
    int k, l;
    double diag_maximum = off_diag_abs(A, &k, &l, n-1);
    start_Jacobi = clock();
    while (diag_maximum > eps && counter < max_counter) {
        diag_maximum = off_diag_abs(A, &k, &l, n-1);
        Jacobi_rotate(A, R, k, l, n-1);
        counter++;
    }
    finish_Jacobi = clock();
    calculation_time_Jacobi = (finish_Jacobi - start_Jacobi)/(double)CLOCKS_PER_SEC;
    // Extract diagonal elements (eigenvalues) and sort them:
    for (int i = 0; i < n-1; i++) { eigenvalues(i) = A(i,i); }
    eigenvalues = sort(eigenvalues);


    // Calculate computation time and eigenvectors with standard Armadillo library:
    clock_t start_arma, finish_arma;
    start_arma = clock();
    mat eigvec;
    vec eigval;
    eig_sym(eigval, eigvec, B);
    finish_arma = clock();
    calculation_time_arma = (finish_arma - start_arma)/(double)CLOCKS_PER_SEC;

    // Write results to outfile:
    ofile << setw(5) << n << "␣␣␣␣";
    ofile << "{" << setprecision(6) << eigenvalues(0) << ",␣";
    ofile << setprecision(6) << eigenvalues(1) << ",␣";
    ofile << setprecision(6) << eigenvalues(2) << "}";
    ofile << setw(12) << setprecision(6) << calculation_time_arma << "␣s.";
    ofile << setw(12) << setprecision(6) << calculation_time_Jacobi << "␣s.";
    ofile << setw(10) << counter << endl;

    }
    ofile.close();
    return 0;
}
```

Listing 5: Program: `Project2_a.py`. It compiles and runs `Project2_a.cpp`.

```
# Program for running Jacobi's rotation algorithm

import os

os.system('g++␣Project2_a.cpp␣-o␣Project2_a.o␣-O2␣-I␣/Users/Henrik/FYS4150\␣-\␣
    Computational\␣physics/armadillo-5.500.2/include␣-lblas␣-llapack')
os.system('./Project2_a.o')
```

Listing 6: Program: `Project2_a_plot_eigenfunctions.cpp` which is compiled and ran by calling `Project2_a_plot_eigenfunctions.py`.

```
/*
**      Project2: a)
**      A program to plot the normalized eigenfunctions.
**      They are compared to exact wave functions.
*/

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <armadillo>
#include <cstring>
#include <time.h>

using namespace std;
using namespace arma;
ofstream ofile;

int main() {

    // Initial variables:
    int n = 350;
    double rho_min, rho_max, h, e;
    double calculation_time_arma;
    string outfilename;

    rho_min = 0.0;
    rho_max = 5.0;
    outfilename = "Wavefunction.txt";

    h = (rho_max - rho_min)/n;
    e = -1.0/(h*h);

    vec d(n-1);
    vec rho(n+1);
    mat A(n-1,n-1);

    for (int i=0; i <= n; i++) { rho(i) = rho_min + i*h; }
    // Now rho(0) = rho_min and rho(n) = rho_max.
    for (int i=0; i < n-1; i++) { d(i) = 2/(h*h) + rho(i+1)*rho(i+1); }
    // Now d(0) is evaluated at rho(1) etc.

    // Initializing matrix A:
    for (int i=0; i < n-1; i++) {
          A(i,i) = d(i);
          if (i != n-2) {
                A(i,i+1) = e;
                A(i+1,i) = e;
          }
    }

    // Calculate computation time and print out eigenvalues to check:
    clock_t start_arma, finish_arma;
    start_arma = clock();
    mat eigvec;
    vec eigval;
    eig_sym(eigval, eigvec, A);
    finish_arma = clock();
    calculation_time_arma = (finish_arma - start_arma)/(double)CLOCKS_PER_SEC;

    cout << endl;
    cout << "Standard armadillo solver. Time: " << calculation_time_arma << " s." <<
        endl;
    for (int i= 0; i < 3; i++) {
        cout << eigval(i) << endl;
    }

    // Eigenfunctions are normed by the use of a simple trapezoidal integration:
    double norm1, norm2, norm3;
    vec V1 = eigvec.col(0);
    vec V2 = eigvec.col(1);
    vec V3 = eigvec.col(2);

    vec U_square_normed1(n-1);
    vec U_square_normed2(n-1);
    vec U_square_normed3(n-1);

    for (int i = 0; i < n-1; i++) {
        double V1_i2 = V1(i)*V1(i);
        double V2_i2 = V2(i)*V2(i);
        double V3_i2 = V3(i)*V3(i);

        norm1 += V1_i2; norm2 += V2_i2; norm3 += V3_i2;
        U_square_normed1(i) = V1_i2;
        U_square_normed2(i) = V2_i2;
        U_square_normed3(i) = V3_i2;
    }
    norm1 *= h; norm2 *= h; norm3 *= h;
    U_square_normed1 /= norm1;
    U_square_normed2 /= norm2;
    U_square_normed3 /= norm3;

    // Normed results are written to file and read by a python script:
    ofile.open(outfilename);
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << "       rho:          u1              u2              u3:" << endl;
    // Loop over all n producing table of time used:
    for (int i=0; i<n-1; i++) {
      double U_val1 = U_square_normed1(i);
      double U_val2 = U_square_normed2(i);
      double U_val3 = U_square_normed3(i);
      ofile << setw(15) << setprecision(8) << rho(i+1);
      ofile << setw(15) << setprecision(8) << U_val1;
      ofile << setw(15) << setprecision(8) << U_val2;
      ofile << setw(15) << setprecision(8) << U_val3 << endl;
    }
    ofile.close();

    return 0;
}
```

Listing 7: Program: `Project2_a_plot_eigenfunctions.py`. It compiles and runs `Project2_a_plot_eigenfunctions.cpp`. This program was used to produce figure 1.

```
# Program for plotting the wave functions found
# with Armadillo library and comparing them to
# exact results.

import os

os.system('g++ Project2_a_plot_eigenfunctions.cpp -o Project2_a_plot_eigenfunctions.o -
    O2 -I /Users/Henrik/FYS4150\ -\ Computational\ physics/armadillo-5.500.2/include -
    lblas -llapack')
os.system('./Project2_a_plot_eigenfunctions.o')

from math import *
import numpy as np
import matplotlib.pyplot as plt

def read_rho_wavefunctions(filename):
    infile = open(filename, 'r')
    # Elements to be read in file:
    rho = []; u1 = []; u2 = []; u3 = [];
    # Read lines except for the first one:
    lines = infile.readlines()[1:]
    for line in lines:
        words = line.split()
        rho.append(float(words[0]))
        u1.append(float(words[1]))
        u2.append(float(words[2]))
        u3.append(float(words[3]))
    infile.close()
    return rho, u1, u2, u3

# Exact results are defined in some functions:
def analytical_ground_state1(rho):
        return 4/sqrt(pi)*rho**2*exp(-rho**2)

def analytical_ground_state2(rho):
        return 8/(3*sqrt(pi))*rho**2*(1.5-rho**2)**2*exp(-rho**2)

def analytical_ground_state3(rho):
        return 8.0/(15*sqrt(pi))*rho**2*(15.0/4-5*rho**2+rho**4)**2*exp(-rho**2)

# Fetching data by a call on read_x_u_v for three different n:
rho, u1, u2, u3 = read_rho_wavefunctions('Wavefunction.txt')
analytical1 = []; analytical2 = []; analytical3 = [];
for i in range(len(rho)):
        rho_val = rho[i]
        analytical1.append(analytical_ground_state1(rho_val))
        analytical2.append(analytical_ground_state2(rho_val))
        analytical3.append(analytical_ground_state3(rho_val))
        #print "Analytical1: ", analytical_ground_state1(rho_val), " Numerical1: ", u1[
            i]

# Plotting commands:
plt.rcParams.update({'font.size': 14})
fig, ax = plt.subplots(1)
ax.plot(rho,u1,'r-',label='$\mid␣u_0(\\rho)␣\mid^2$,␣numerical')
ax.plot(rho,u2,'b-',label='$\mid␣u_1(\\rho)␣\mid^2$,␣numerical')
ax.plot(rho,u3,'g-',label='$\mid␣u_2(\\rho)␣\mid^2$,␣numerical')
ax.plot(rho,analytical1,'r+',label='$\mid␣u_0(\\rho)␣\mid^2$,␣analytical')
ax.plot(rho,analytical2,'b+',label='$\mid␣u_1(\\rho)␣\mid^2$,␣analytical')
ax.plot(rho,analytical3,'g+',label='$\mid␣u_2(\\rho)␣\mid^2$,␣analytical')
ax.set_xlabel('Radial␣coordinate␣$\\rho$')
ax.set_ylabel('Radial␣wavefunction:␣$\mid␣u(\\rho)␣\mid^2$')
ax.legend(loc='upper␣right',fancybox='True')
ax.set_title('Radial␣wavefunction.')
ax.grid()
plt.show()
```

Listing 8: Program: `Project2_c_eigenvalues.cpp` which is compiled and ran by calling `Project2_c_eigenvalues.py`.

```
/*
**      Project2: c)
**      A program to print out the
**      eigenvalues for comparison.
*/

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <armadillo>
#include <cstring>
```

```cpp
#include <time.h>

using namespace std;
using namespace arma;
ofstream ofile;

// Approximate formula derived in reference article:
double article_eigenvalue_approximate(int m, double omega_r) {
    double epsilon = 3.0/2.0*pow(omega_r/2.0, 2.0/3.0) + sqrt(3.0)*omega_r*(m+0.5);
    return 2.0*epsilon;
}

int main() {

    // Initial variables: ///////
    // rho_max and n are meant to be adjusted.
    int n = 350;
    double rho_min, rho_max, h, e;
    double calculation_time_arma;
    double omega_r = 5.0;
    string outfilename;

    rho_min = 0.0;
    rho_max = 2.0;
    /////////////////////////////

    h = (rho_max - rho_min)/n;
    e = -1.0/(h*h);

    vec d(n+1);
    vec rho(n+1);
    mat A(n-1,n-1);

    for (int i=0; i <= n; i++) { rho(i) = rho_min + i*h; }
    // Now rho(0) = rho_min and rho(n) = rho_max.
    for (int i=0; i < n-1; i++) { d(i) = 2.0/(h*h) + omega_r*omega_r*rho(i+1)*rho(i+1)
        + 1.0/rho(i+1); }
    // Now d(0) is evaluated at rho(1) etc.

    // Initiating matrix A:
    for (int i=0; i < n-1; i++) {
      A(i,i) = d(i);
      if (i != n-2) {
        A(i,i+1) = e;
        A(i+1,i) = e;
      }
    }

    mat eigvec;
    vec eigval;
    eig_sym(eigval, eigvec, A);
    double temp_eigenval;

    // Print out information to screen:
    cout << endl;
    cout << "Ground␣state␣information␣with␣oscillator␣frequency:␣" << omega_r << endl;
    temp_eigenval = article_eigenvalue_approximate(0,omega_r);
    cout << "Eigenvalue␣(groundtate)␣found␣numerically:␣" << eigval(0) << endl;
    cout << "Article␣formula:␣" << temp_eigenval << endl;
    cout << "Relative␣error:␣" << (eigval(0)-temp_eigenval)/temp_eigenval << endl;
    cout << endl;

    return 0;
}
```

Listing 9: Program: `Project2_c_eigenvalues.py`. It compiles and runs `Project2_c_eigenvalues.cpp`.

```
# Program for running Jacobi's rotation algorithm

import os

os.system('g++␣Project2_c_eigenvalues.cpp␣-o␣Project2_c_eigenvalues.o␣-O2␣-I␣/Users/
    Henrik/FYS4150\␣-\␣Computational\␣physics/armadillo-5.500.2/include␣-lblas␣-llapack
    ')
os.system('./Project2_c_eigenvalues.o')
```

Listing 10: Program: `Project2_d.cpp` which is compiled and ran by calling `Project2_d.py`.

```
/*
**      Project2: d)
**      A program to plot the
**      normalized eigenfunctions of the two
**      interacting electron problem.
**      n and omega_r are meant to be adjusted
*/

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <armadillo>
#include <cstring>
#include <time.h>

using namespace std;
using namespace arma;
ofstream ofile;

int main() {

    // Initial variables: ///////
    int n = 350;
    double rho_min, rho_max, h, e;
    double calculation_time_arma;
    double omega_r = 5.0;
    string outfilename;

    rho_min = 0.0;
    rho_max = 2.0;
    outfilename = "Wavefunction_relative.txt";
    /////////////////////////////

    h = (rho_max - rho_min)/n;
    e = -1.0/(h*h);

    vec d(n+1);
    vec rho(n+1);
    mat A(n-1,n-1);

    for (int i=0; i <= n; i++) { rho(i) = rho_min + i*h; }
    // Now rho(0) = rho_min and rho(n) = rho_max.
    for (int i=0; i < n-1; i++) { d(i) = 2/(h*h) + omega_r*omega_r*rho(i+1)*rho(i+1) +
        1.0/rho(i+1); }
    // Now d(0) is evaluated at rho(1) etc.

    // Initiating matrix A:
    for (int i=0; i < n-1; i++) {
      A(i,i) = d(i);
      if (i != n-2) {
        A(i,i+1) = e;
        A(i+1,i) = e;
      }
    }

    // Calculating time used on calculation.
    // Using Armadillo library functions for solving the eigenvalue problem.
    clock_t start_arma, finish_arma;
    start_arma = clock();
    mat eigvec;
    vec eigval;
    eig_sym(eigval, eigvec, A);
    finish_arma = clock();
    calculation_time_arma = (finish_arma - start_arma)/(double)CLOCKS_PER_SEC;

    // Printing out the three lowest eigenvalues to see:
    cout << endl;
    cout << "Standard␣armadillo␣solver.␣Time:␣" << calculation_time_arma << "␣s." <<
        endl;
    for (int i= 0; i < 3; i++) {
        cout << eigval(i) << endl;
    }

    // Normalizing eigenvectors with a simple trapezoidal rule:
    double norm1, norm2, norm3;
    vec V1 = eigvec.col(0);
    vec V2 = eigvec.col(1);
    vec V3 = eigvec.col(2);

    vec U_square_normed1(n-1);
    vec U_square_normed2(n-1);
    vec U_square_normed3(n-1);

    for (int i = 0; i < n-1; i++) {
        double V1_i2 = V1(i)*V1(i);
        double V2_i2 = V2(i)*V2(i);
        double V3_i2 = V3(i)*V3(i);

        norm1 += V1_i2; norm2 += V2_i2; norm3 += V3_i2;
        U_square_normed1(i) = V1_i2;
        U_square_normed2(i) = V2_i2;
        U_square_normed3(i) = V3_i2;
    }
    norm1 *= h; norm2 *= h; norm3 *= h;
    U_square_normed1 /= norm1;
    U_square_normed2 /= norm2;
    U_square_normed3 /= norm3;

    // Writing all the normalized results to a txt file and plot them
    // with a python script.
    ofile.open(outfilename);
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << "␣␣␣␣␣␣␣rho:␣␣␣␣␣␣␣␣␣u1␣␣␣␣␣␣␣␣␣␣␣␣␣u2␣␣␣␣␣␣␣␣␣␣␣␣␣u3:" << endl;
    // Loop over all n producing table of time used:
    for (int i=0; i<n-1; i++) {
      double U_val1 = U_square_normed1(i);
      double U_val2 = U_square_normed2(i);
      double U_val3 = U_square_normed3(i);
      ofile << setw(15) << setprecision(8) << rho(i+1);
      ofile << setw(15) << setprecision(8) << U_val1;
      ofile << setw(15) << setprecision(8) << U_val2;
      ofile << setw(15) << setprecision(8) << U_val3 << endl;
    }
    ofile.close();

    return 0;
}
```

Listing 11: Program: `Project2_d.py`. It compiles and runs `Project2_d.cpp`. This program was used to produce the figures 2 and 3.

```
# Program to compile and run Project2_d.cpp

import os
```

```
os.system('g++␣Project2_d.cpp␣-o␣Project2_d.o␣-O2␣-I␣/Users/Henrik/FYS4150\␣-\␣
    Computational\␣physics/armadillo-5.500.2/include␣-lblas␣-llapack')
os.system('./Project2_d.o')

from math import *
import numpy as np
import matplotlib.pyplot as plt
omega_r = 5.0

def read_rho_wavefunctions(filename):
    infile = open(filename, 'r')
    # Elements to be read in file:
    rho = []; u1 = []; u2 = []; u3 = [];
    # Read lines except for the first one:
    lines = infile.readlines()[1:]
    for line in lines:
        words = line.split()
        rho.append(float(words[0]))
        u1.append(float(words[1]))
        u2.append(float(words[2]))
        u3.append(float(words[3]))
    infile.close()
    return rho, u1, u2, u3

# Fetching data by a call on read_x_u_v for three different n:
rho, u1, u2, u3 = read_rho_wavefunctions('Wavefunction_relative.txt')

# Plotting commands to look at the wave functions:
plt.rcParams.update({'font.size': 14})
fig, ax = plt.subplots(1)
ax.plot(rho,u1,'r-',label='$\mid␣u_0(\\rho)␣\mid^2$')
ax.plot(rho,u2,'b-',label='$\mid␣u_1(\\rho)␣\mid^2$')
ax.plot(rho,u3,'g-',label='$\mid␣u_2(\\rho)␣\mid^2$')
ax.set_xlabel('Radial␣coordinate␣$\\rho$')
ax.set_ylabel('Relative␣radial␣wavefunction:␣$\mid␣u(\\rho)␣\mid^2$')
ax.legend(loc='upper␣right',fancybox='True')
ax.set_title('Relative␣radial␣wave␣function␣with␣$\omega_r␣=$␣%.2f.' % omega_r)
ax.grid()
plt.show()
```

Listing 12: Program: `Project2_f_Lanczos.cpp` which is compiled and ran by calling `Project2_f_Lanczos.py`.

```
/*
**      Project2: f)
**      A first implementation of Lanczos' algorithm.
**      The function Lanczos() takes in matrix A and
**      transforms it to a tridiagonal matrix which is
**      returned. If one puts q1(0) = 1 (and the rest equal
**      to zero), this produces basically the same matrix as
**      with Householder's algorithm.
*/

#include <iostream>
#include <cmath>
#include <armadillo>
#include <fstream>
#include <iomanip>
#include <cstring>

using namespace std;
using namespace arma;
ofstream ofile;

// Computes and returns the 2-norm of a vector v:
double norm_by_hand(vec v, int n) {
    double sum_norm = 0.0;
    for (int i=0; i < n; i++) {
        sum_norm += v(i)*v(i);
    }
    sum_norm = sqrt(sum_norm);
    return sum_norm;
}

// Transform matrix to tridiagonal form with Lanczos'
// iterative algorithm and later extract its eigenvalues by
// some other method.
void Lanczos(mat & A, int n) {
    // tolerance in beta values:
    double btol = 1E-5;
    // initiate vectors and matrices used in the loop:
    int kmax = n;
    int k = 0;
    vec alpha = zeros(kmax);
    vec beta = zeros(kmax);
    vec q_k = zeros(n);
    vec q1 = zeros(n);
    //q1(0) = 1;
    // Generate random normed vector:
    q1 = randu<vec>(n);
    q1 /= norm_by_hand(q1,n);
    vec r = q1;
    double b = 1;

    // Loop over k:
    while (b > btol && k < kmax) {
        k += 1;
        vec qkm1 = q_k;
        q_k = r/b;
        vec Aqk = A*q_k;
        // Update alpha element:
        alpha(k-1) = dot(trans(q_k),Aqk);
        r = Aqk - q_k*alpha(k-1) - qkm1*b;
        b = norm_by_hand(r,n);
        // Update beta element:
        beta(k-1) = b;
    }

    // Overwrite matrix A with the newly computed tridiagonal one:
    A = zeros(n,n);
    for (int i=0; i < n; i++) {
        A(i,i) = alpha(i);
        if (i != n-1) {
            A(i+1,i) = beta(i);
            A(i,i+1) = beta(i);
        }
    }
    return;
}

int main() {
    // Initial variables:
    int n = 50;
    string outfilename;
    mat A(n,n, fill::randu);
    // Produce an initially symmetric matrix:
    mat B = trans(A);
    A = 0.5*(B+A);

    // Calculate eigenvalues with Armadillo library directly on A:
    mat eigvec0;
    vec eigval0;
    eig_sym(eigval0, eigvec0, A);

    // Make A tridiagonal to check if the matrix has
    // eigenvalues close to that of the original matrix:
    Lanczos(A,n);
    mat eigvec;
    vec eigval;
    eig_sym(eigval, eigvec, A);

    outfilename = "Table_of_results_Lanczos.txt";
    ofile.open(outfilename);
    ofile << setiosflags(ios::showpoint | ios::uppercase);

    // Write results to txt file:
    ofile << "Eigenvalues␣found␣with␣Armadillo:␣" << endl;
    ofile << "On␣matrix␣A:␣␣␣␣On␣matrix␣T:␣␣␣Relative␣difference:" << endl;
    for (int i=0; i < n; i++) {
        ofile << setprecision(6) << setw(10) << eigval0(i) << "␣␣␣␣␣␣␣";
        ofile << setprecision(6) << setw(10) << eigval(i) << "␣␣␣␣␣␣␣␣␣";
        ofile << setw(10) << fabs((eigval0(i) - eigval(i))/eigval0(i)) << endl;
    }
    ofile.close();
    return 0;
}
```

Listing 13: Program: `Project2_f_Lanczos.py`. It compiles and runs `Project2_f_Lanczos.cpp`. This produces the output file `Table_of_results_Lanczos.txt`.

```
# Program for running Project2_f_Lanczos.cpp

import os

os.system('g++␣Project2_f_Lanczos.cpp␣-o␣Project2_f_Lanczos.o␣-O2␣-I␣/Users/Henrik/
    FYS4150\␣-\␣Computational\␣physics/armadillo-5.500.2/include␣-lblas␣-llapack')
os.system('./Project2_f_Lanczos.o')
```