Computational Physics - Project 2

Johannes Scheller, Vincent Noculak, Lukas Powalla

October 4, 2015

Contents

1 Introduction And Motivation

In many fields of both mathematics and physics, we often come to the point that we have to solve so-called eigenvalue problems, which are equations of the form $\hat{A} \cdot \hat{v} = \lambda \hat{v}$, where $\hat{A}$ is a matrix of dimension $n \times n$ and v is a vector of dimension n. Equations of this kind occur not only in linear algebra, but also in mechanics and quantum mechanics and will also be a major part of this report. In this project, we are going to rewrite the Schrödinger's equation of one and two electrons in a harmonic oscillator potential in the form of an eigenvalue problem and solve it numerically by implementing Jacobi's method, an algorithm that can be used to solve any eigenvalue problem. This algorithm is based on simple rotations of a given matrix to diagonalize it. We will show how this algorithm works, set it up on our own using C++ and use this program to obtain the eigenvalues of the Schrödinger's equation of electrons in an harmonic oscillator potential that we are going to rewrite as an eigenvalue problem.

2 Theory

2.1 Rewriting Schrödinger's equation as eigenvalue problem

2.1.1 One electron in a harmonic oscillator potential

2.1.2 Two interacting electrons in a harmonic oscillator potential

2.2 Jacobi's method

3 Execution

3.1 Implementing the algorithm

Our main goal in this project was to implement our own algorithm for Jacobi's method to compute the eigenvalues of the equation discussed above. This was done in the source files that are found under XXX. First, our program asks for the desired number of steps n and sets up the matrix and the vectors for the eigenvalue equation as described in section **??**, as well as a matrix $\hat{R}$ which is used for to compute the eigenvectors respectively the eigenfunctions. Then, it starts the actual Jacobi's method: It looks for the absolutely highest non-diagonal element of the matrix in the function *maxoffdiag*, and, while this is higher than the chosen tolerance ϵ, it performs a rotation around the line and the column of this element in the function *rotation*. This function obtains the values of τ, $\sin(\theta)$, $\cos(\theta)$ and $\tan(\theta)$ by using the formulas that we introduced in section **??**.

```
//The Jacobi algorithm gets executed
while ( fabs(max_offdiag) > epsilon && iterations < max_number_iterations )
{

        rotate ( A, R, k, l, n );
        max_offdiag = maxoffdiag ( A, &k, &l, n );
        iterations++;

}
// Function to find the maximum matrix element
double maxoffdiag ( double ** A, int * k, int * l, int n )
{
        double max = 0.0;
        for ( int i = 0; i < n-1; i++ )
        {
                for ( int j = i + 1; j < n-1; j++ )
                {
                        if ( fabs(A[i][j]) > max )
                        {
                                max = fabs(A[i][j]);
                                *l = i;
                                *k = j;
                        }
                }
        }
        return max;
}
```

```
// Function to find the values of cos and sin and then to rotate
void rotate ( double ** A, double ** R, int k, int l, int n )
{
        double s, c;
        if ( A[k][l] != 0.0 )
        {
                double t, tau;
                tau = (A[l][l] - A[k][k])/(2*A[k][l]);
                if ( tau > 0 )
                {
                        t = 1.0/(tau + sqrt(1.0 + tau*tau));
                }
                else
                {
                        t = -1.0/( -tau + sqrt(1.0 + tau*tau));
                }
                c = 1/sqrt(1+t*t);
                s = c*t;
        }
        else
        {
                c = 1.0;
                s = 0.0;
        }
        double a_kk, a_ll, a_ik, a_il, r_ik, r_il;
        a_kk = A[k][k];
        a_ll = A[l][l];
        // changing the matrix elements with indices k and l
        A[k][k] = c*c*a_kk - 2.0*c*s*A[k][l] + s*s*a_ll;
        A[l][l] = s*s*a_kk + 2.0*c*s*A[k][l] + c*c*a_ll;
        A[k][l] = 0.0;
        A[l][k] = 0.0;
        // and then we change the remaining elements
        for ( int i = 0; i < n-1; i++ )
        {
                if ( i != k && i != l )
                {
                        a_ik = A[i][k];
                        a_il = A[i][l];
                        A[i][k] = c*a_ik - s*a_il;
                        A[k][i] = A[i][k];
                        A[i][l] = c*a_il + s*a_ik;
                        A[l][i] = A[i][l];
                }
                //Compute new eigenvectors in its matrix
                r_ik = R[i][k];
                r_il = R[i][l];
                R[i][k] = c*r_ik - s*r_il;
                R[i][l] = c*r_il + s*r_ik;
        }
        return;
}
```

As a simple test case for our algorithm, we took a 2×2 matrix which it solved correctly within 2 steps. This indicates that the program is working correctly, a fact that is highlighted by the correct results for the eigenvalues that we obtained during this project. At the end, our program gives the user the possibility to print out all eigenvalues, a specific one or only the first three eigenvalues.

3.2 Setting and Testing of Parameters

In both cases, whether we deal with only one particle or with two, we have three parameters to be set, resulting in two degrees of freedom that have an effect on the accuracy of our results. We tested several set-ups for those three parameters for the one particle case to obtain the best and took slight changes to adjust it for the case with two interacting particles.

The first and most obvious parameter is n, the number of grid points we use. Using a higher value of n, we gain more precision as the step length decreases, but at the same time, we will end up with a larger matrix that needs more memory (proportional to n^2). Most important, the number of similarity transformations needed to calculate the eigenvalues goes like n^3, leading to a very long computation time for large matrices. The highest possible value we used for n was 1000, resulting in more than 45 minutes of computation time! Altering n had the most effect on our results, so we decided to take a relatively high value for this parameter.

The second parameter that we can alter is ρ_{max}, the maximum value of ρ. In theory, this value should be infinite, which is just not possible for this numeric solution. In our case, the higher an eigenvalue is, the more its calculation depends on the choice of ρ_{max}. Therefore the challenge was to set this parameter to a value which resulted in stable and consistent results for the first three eigenvalues without being to high, as a higher value would also increase our step length h if we don't change n accordingly. Choosing the right value for this parameter wasn't easy as a higher value would affect the lowest eigenvalue to become more imprecise due to the increasing step length, but would at the same time improve our results for the higher eigenvalues. This is the case since the eigenfunctions of higher eigenvalues extend to a larger value of ρ and are therefore affected more by a relatively small value of ρ_{max}.

The last degree of freedom is to set the tolerance for the non-diagonal matrix elements that are supposed to become zero. This value determines implicitly how many similarity transformations are being operated until the non-diagonal elements are considered zero. A smaller value can lead to higher precision in the eigenvalues, but will at the same time increase the computation time again. We found out that changing this value had a comparably small effect on the final results: We tried the values of $\epsilon = 10^{-8}$ and $\epsilon = 10^{-10}$, which led to no difference at all in five leading digits for the five lowest eigenvalues! Therefore, we decided to choose $\epsilon = 10^{-8}$, which obviously needed less computation time.

As said above, we tested different set-ups with different values of n and ρ_{max} with the results shown in tab. 1. As we wanted a precision of three leading digits for the three lowest eigenvalues, we decided to use the set-up with $n = 400$, $\rho_{max} = 6$ and $\epsilon = 10^{-8}$ for our final results of the one particle case, which seemed to be a good compromise between precision and computation time and led to the desired results. For the two particle case, we had to perform some changes to obtain stable results, which are explained in section **??**.

Table 1: Comparing different set-ups of the parameters n and ρ_{max} for the one particle case

$\epsilon = 10^{-8}$	1. Eigenvalue	2. Eigenvalue	3. Eigenvalue	4. Eigenvalue	5. Eigenvalue
$n = 200$, $\rho_{max} = 5$	2.9998	6.9990	10.9979	15.0024	19.0751
$n = 200$, $\rho_{max} = 8$	2.9995	6.9975	10.9939	14.9887	18.9819
$n = 200$, $\rho_{max} = 10$	2.9992	6.9961	10.9905	14.9823	18.9717
$n = 500$, $\rho_{max} = 5$	2.99997	6.9999	10.9998	15.0055	19.0756
$n = 500$, $\rho_{max} = 8$	2.99992	6.9996	10.9990	14.9982	18.9971
$n = 500$, $\rho_{max} = 10$	2.99987	6.9994	10.9985	14.9972	18.9955
$n = 1000$, $\rho_{max} = 5$	2.99999	6.99996	11.0001	15.0059	19.0746
$n = 1000$, $\rho_{max} = 8$	2.99998	6.9999	10.9998	14.9995	18.9993
$n = 1000$, $\rho_{max} = 10$	2.99997	6.99984	10.9996	14.9993	18.9989
Exact values	3	7	11	15	19

3.3 Results

4 Comparison and discussion of the results