FYS3150
Project 2 - The Jacobi strikes back

Hugounet, Antoine & Villeneuve, Ethel

September 2017
University of Oslo
`https://github.com/kryzar/Vega.git`

Abstract

This project aims to solve the Schrödinger equation and find the associate wave function. We have an harmonic oscillator with one electron or two with the Coulomb interaction and rewrite the equation as an eigenvalue problem by scalling it and discretizing it. We will use the Jacobi method of diagonalization to solve the problem, by finding the eigenvalues of our discrete problem. Those eigenvalues correspond to the energies in the Schrödinger equation. With those results, we could compute the corresponding eigenvectors, which represent the wave function.

The matrix we come up with is a tridiagonal symmetric matrix and as a consequence, the program we write will only compute symmetric matrices. The rate of convergence to the diagonal matrix is quite low, but this algorithm has the advantage of being simple to implement. We added unit tests and one invariant test to invalidate the results.
The program writes the data to a file with additional information including the operation time and the number of symmetric transformations. It is thus very convenient to read the data as a human being or as a plot program.

Contents

Introduction

If you like physics and computer sciences, and that you don't like solving equations, then scaling and discretizing is always a good idea. Once again it enables us to apply a general algorithm - a standard eigenvalue solver algorithm - to a very specific problem. If we do well, a great virtue of this project is that we'll be able to use the same code many times, since we develop a general eigenvalue solver for symmetric matrices. The first chapter of this report shows how we get from the original Schrödinger equation to a standard problem. Then we translate the math into a program, and the results are in the third chapter.

Chapter 1

Theory

Two electrons are moving in a three-dimensional harmonic oscillator well. We are looking for the solution of the Schrödinger equation in this case. First, we are interested in the case where we have only one electron in a harmonic oscillator and we will then move to the case with two electrons without and with interaction.

1.1 One electron

1.1.1 The Schrödinger equation

The time-independent Schrödinger equation is $H\psi = E\psi$, ψ being the wave function, H the Hamiltoninan operator and E the energy of the state. In our case we assume spherical symmetry. We have $H = \frac{-\hbar}{2m}\nabla^2 + V(r)$ and the Schrödinger equation reads :

$$\frac{-\hbar}{2m}\nabla^2 + V(r)\psi = E\psi$$

We want the radial part of the Schrödinger equation. We take then $\psi = R(r)$

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

with $V(r)$ the harmonic oscillator potential, l the second quantum number (the orbital momentum of the electron) and $R(r)$ the radial part of ψ.

$V(r) = \frac{1}{2}kr^2$ with $k = m\omega^2$ (ω the oscillator frequency) and $E_{nl} = \hbar\omega(2n + l + \frac{3}{2})$ with $n, l \in \mathbb{N}$. We set $R(r) = \frac{u(r)}{r}$. The radial part of the Schrödinger equation for one electron is now :

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

We have the following boundary conditions : $u(0) = 0$ and $u(\infty) = 0$ so we can define ρ as a dimensionless variable $\rho = \frac{r}{\alpha}$ with α a constant :

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

and

$$V(\rho) = \frac{1}{2}k\alpha^2\rho^2$$

We are interested in the ground state so we take $l = 0$. The equation can be then written as

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

$$\implies -\frac{d^2}{d\rho^2}u(\rho) + \frac{mk}{\hbar^2}\alpha^4\rho^2 u(\rho) = \frac{2m\alpha^2}{\hbar^2}Eu(\rho)$$

We fix α so that $\alpha = \left(\frac{\hbar^2}{mk}\right)^{\frac{1}{4}}$ and we define $\lambda = \frac{2m\alpha^2}{\hbar^2}E$. We can now write the one-electron Schrödinger equation as

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

1.1.2 Discretized formula

We want to resolve this equation numerically. To do this, we need to discretize the Schrödinger equation to compute the eigenvalues.

The second derivative of u is given by

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

where h is our step, $h = \frac{\rho_N - \rho_0}{N}$ with N the number of mesh points, $\rho_N = \rho_{max}$ and $\rho_0 = \rho_{min}$. To computize the function, we discretize the values : we define ρ_i as the value of ρ at a point i with $i = \overline{1, N}$. Then, $\rho_i = \rho_0 + ih$. The Schrödinger equation for a value ρ_i is

$$-\frac{u(\rho_i + h) - 2u(\rho_i) + u(\rho_i - h)}{h^2} + \rho_i^2 u(\rho_i) = \lambda u(\rho_i)$$

$$\implies -\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \rho_i^2 u_i = \lambda u_i \tag{3}$$

and as we have the harmonic oscillator potential $V_i = \rho_i^2$:

$$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + V_i u_i = \lambda u_i$$

If we write this equation in matrix form, we end up with

$$\begin{bmatrix} \frac{2}{h^2} + V_1 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2} + V_2 & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \dots & 0 & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2} + V_{N-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2} + V_{N-1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N-1} \end{bmatrix} = \lambda \begin{bmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N-1} \end{bmatrix} \tag{4}$$

and $u_0 = u_N = 0$.

1.2 Two electrons

We want now to express the Schrödinger equation for two electrons in a harmonic oscillator well.

1.2.1 Non-interacting case

We have from (1)

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

with $E^{(1)}$ the energy with one electron. Then, if we have two electrons, the equation simply becomes

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

with $E^{(2)}$ the energy for two electrons, without interaction.
We introduce the relative coordinate : $\mathbf{r} = \mathbf{r_1} - \mathbf{r_2}$, the center-of-mass coordinates $\mathbf{R} = \frac{1}{2}(\mathbf{r_1} + \mathbf{r_2})$ and the total energy given by the sum of the relative energy E_r and the center-of-mass energy E_R : $E^{(2)} = E_r + E_R$. With these new coordinates, the Schrödinger equation can be written as

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

We can separate the wave function into the product of two functions : $u(r, R) = \psi(r)\phi(R)$. The Schrödinger equation reads :

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} - \frac{\hbar^2}{4m}\frac{d^2}{dR^2} + \frac{1}{4}kr^2 + kR^2\right)\psi(r)\phi(R) = (E_r + E_R)\psi(r)\phi(R) \tag{5}$$

1.2.2 Interacting case

We add the repulsive Coulomb interaction between the two electrons to the last equation: $V(r_1, r_2) = \frac{\beta e^2}{|\mathbf{r_1} - \mathbf{r_2}|} = \frac{\beta e^2}{r}$ with $\beta e^2 = 1.44$ eVnm. Only the R-dependent equation will not change, and we can write the r-dependent Schrödinger equation as

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} + \frac{1}{4}kr^2 + \frac{\beta e^2}{r}\right)\psi(r) = E_r\psi(r) \tag{6}$$

As we did before, we set $\rho = \frac{r}{\alpha}$. By replacing r with $\alpha\rho$ and multiplying each side by $\frac{m\alpha^2}{\hbar^2}$ we obtain

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4\rho^2\psi(\rho) + \frac{m\alpha\beta e^2}{\rho\hbar^2}\psi(\rho) = \frac{m\alpha^2}{\hbar^2}E_r\psi(\rho)$$

We define $\omega_r^2 = \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4$ –where ω is a parameter which express the stregth of the oscillator potential–, fix the constant α so that $\alpha = \frac{\hbar^2}{m\beta e^2}$, and set $\lambda = \frac{m\alpha^2}{\hbar^2}E$. The Schrödinger equation becomes

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \left(\omega_r^2\rho^2 + \frac{1}{\rho}\right)\psi(\rho) = \lambda\psi(\rho) \tag{7}$$

As the equations (7) and (2) look alike, we will use the same algorithm to solve the Schrödinger equation in both cases, only changing the value of V_i ($V = \rho^2$ for the non-interacting case and $V = \omega^2\rho^2 + \frac{1}{\rho}$ for the interacting case). ψ is a real-valued function. If we discretize it, we discretize the other parameters as well and get the following result as previously :

$$\mathbf{A}\psi(\rho) = \lambda\psi(\rho) \tag{8}$$

1.3 Jacobi method

The Jacobi's method is an algorithm to transform a matrix into its own diagonal matrix. In an eigenvalue problem like ours, it will help us to find the eigenvalues which will be the diagonal elements of the resulting matrix.

We have the following matrix

$$\mathbf{A} = \begin{bmatrix} \frac{2}{h^2} + V_1 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2} + V_2 & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \dots & 0 & \dots & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2} + V_{N-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2} + V_{N-1} \end{bmatrix}$$

The goal is to have something like

$$\mathbf{B} = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & \dots & 0 \\ 0 & 0 & \lambda_3 & 0 & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \dots & 0 & \lambda_{N-2} & 0 \\ 0 & 0 & 0 & \dots & \dots & 0 & \lambda_{N-1} \end{bmatrix}$$

with $\mathbf{B}$ being $\mathbf{A}$ transformed.

First we are looking for the largest non-diagonal element of the matrix $\mathbf{A}$ which will be the element a_{kl}. The transformation matrix is defined by

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

with $s_{kk} = s_{ll} = cos\theta$; $s_{kl} = -s_{lk} = -sin\theta$; $s_{ii} = 1$ for $i \neq k$, $i \neq l$. With the transformation $\mathbf{B} = \mathbf{S}^T\mathbf{AS}$, we have :

$$\begin{aligned}
b_{ii} &= a_{ii}; i \neq k, i \neq l \\
b_{ik} &= a_{ik}cos\theta - a_{il}sin\theta; i \neq k, i \neq l \\
b_{il} &= a_{il}cos\theta + a_{ik}sin\theta; i \neq k, i \neq l \\
b_{kk} &= a_{kk}cos^2\theta - 2a_{kl}cos\theta sin\theta + a_{ll}sin^2\theta \\
b_{ll} &= a_{l}cos^2\theta - 2a_{kl}cos\theta sin\theta + a_{kk}sin^2\theta \\
b_{kl} &= (a_{kk} - a_{ll})cos\theta sin\theta + a_{kl}(cos^2\theta - sin^2\theta)
\end{aligned}$$

We then look for $\cos\theta$ and $\sin\theta$ such that $b_{kl} = 0$ to find $\mathbf{S}$. We repeat the transformations on $\mathbf{B}$ until it becomes our diagonal matrix, each time with a different $\mathbf{S}$. Indeed, the largest non-diagonal element is on line k and changes and we make it 0 one by one. The problem is that the algorithm does not prevent a 0 non-diagonal elements from being assigned a real non-zero value again. This has the consequence of making the convergence rate of the algorithm very slow, and not unpredictable.

Chapter 2

Implementation

2.1 Jacobi algorithm

2.1.1 jacobi.cpp

The algorithm is `jacobi.cpp`. We used several functions in order to make the code cleaner ; way more that it was in project one. The idea is to divide the process in many small auxiliary functions `ft_function` so to read the algorithm very easily and to make it shorter. As an basic instance, `ft_nondiago_largest` is used to find the largest non-diagonal element.

That being said, the process is pretty straight forward. We transform the largest non-diagonal element to 0 and we do so for each non-diagonal element. Once the largest non-diagonal element is found, we use `ft_find_trigo` to compute $\cos\theta$ and $\sin\theta$ such that $a_{kl} = 0$. Then we change the elements of the matrix and of the eigenmatrix[1] into the corresponding values with the function `ft_rotate`. We go over the loop again until we almost have a diagonal matrix : the non-diagonal element would not surely be 0 because of the numerical precision and we want their absolute value to be within a small ϵ, typically 10^{-8} for a double type.

At the end, we return a sorted dynamic array with the eigenvalues using `ft_diago_to_array`, a function which exports the diagonal elements of the matrice to a sorted dynamic and native `C++` array. We write some data to a file using `ft_cosmetics`, including : the operation time, the eigenvalues and the number of iterations.

Remark 2.1. *The corresponding small functions are here : https://github.com/kryzar/Vega/tree/master/Program/Program/jacobi%20functions. All the descriptions are in the header file.*

The exit codes for this function are :

- `exit(1)` : unit tests failed
- `exit(2)` : the invariant test is not verified

[1] The matrix of which the columns are the eigenvectors.

2.1.2 main.cpp

The main function takes four command lines arguments :

- `argv[1]` : mode [string], "interactive" or "noninteractive"
- `argv[2]` : n [int], number of meshpoints
- `argv[3]` : rho_max [double]
- `argv[4]` : omega [double], only in interactive mode

You can manually use `ft_tests` in `main` to run some tests that will output in a special file. You can also use `ft_arma_compar` the same way to compare our algorithm with Armadillo yourself.

The matrices are initiated with one function for each mode : `ft_(non)interactive_init` of which role is to set the good physical elements in the matrix. Before that, we test the good number of arguments and the good value for the mode with `ft_precautions`. And performance is not the main criteria for this program. You can also notice that `ft_cosmetics`, which writes data to a file, is overloaded for each different mode (interactive or noninteractive).

The exit codes for `main` are :

- `exit(10)` : you chose the wrong mode in a command line argument
- `exit(11)` : you did not input 4 command line arguments in non interactive mode as required
- `exit(12)` : you did not input 5 command line arguments in interactive mode as required

Remark 2.2. *We give a description of each function in the header files as well.*

Remark 2.3. *There are some basic tests that are ready to be ran in the function `ft_tests`. Just change the path to the output file and the results will go to this file. You just have to call this function in `main`.*

2.2 Unit tests

2.2.1 Invariant

This algorithm has been made very secure. We know that the Frobenius norm is invariant under the unitary transformations, and we choose to test it at each call of the function, using `ft_invariant` and the armadillo function to calculate the Frobenius norm. This makes the process a bit slow, but more reliable at the same time. If this norm is preserved, then `ft_cosmetics` prints it in the results file to tell the user "It's okay, you can use this function to get your Nobel prize". If it's not, `ft_invariant` aborts the process.

The trick here is to understand that our matrix is not completely diagonal[2] and that the norm will not be perfectly constant but will have a tiny evolution instead. We coded a function `ft_numerical_equality` to check that for two real numbers x and y, we always have $|x - y| < \epsilon$ for some tiny and lovely ϵ.

2.2.2 Catch unit tests

The invariant verification is already a strong security, but some unit tests are never a bad thing. We have some properties about transformations on vectors and we can use them to test the algorithm. We know that a unitary transformation preserves the inner product :
Let a basis of vectors $\mathbf{v}_i$,

$$\mathbf{v}_i = \begin{bmatrix} v_{i1} \\ \dots \\ \dots \\ v_{in} \end{bmatrix}.$$

A unitary transformation $\mathbf{U}$ is defined by $\mathbf{U}^\dagger\mathbf{U} = 1$. It preserves the inner product : the inner product of two vectors before the transformation is the equal to their inner product after the transformation. Mathematically, $\langle \mathbf{v}_i | \mathbf{v}_j \rangle = \langle \mathbf{w}_i | \mathbf{w}_j \rangle$ with $\mathbf{w}_i = \mathbf{U}\mathbf{v}_i$:

$$\langle \mathbf{w}_i | \mathbf{w}_j \rangle = \langle \mathbf{U}\mathbf{v}_i | \mathbf{U}\mathbf{v}_j \rangle = \sum (\mathbf{U}\mathbf{v}_i)^\dagger \mathbf{U}\mathbf{v}_j = \sum \mathbf{v}_i^* \mathbf{U}^\dagger \mathbf{U}\mathbf{v}_j$$

As the transformation is unitary, $\mathbf{U}^\dagger\mathbf{U} = 1$. So,

$$\langle \mathbf{w}_i | \mathbf{w}_j \rangle = \sum \mathbf{v}_i^* \mathbf{v}_j = \langle \mathbf{v}_i | \mathbf{v}_j \rangle$$

So, a unitary transformation preserves the inner product.

A second property of a unitary transformation is that it preserves orthogonality :
Let's assume $\mathbf{v}_i$ to be an orthogonal basis of vectors, $\mathbf{v}_j^T \mathbf{v}_i = \delta_{ij}$. A unitary tranformation preserves its orthgonality, which means $\mathbf{v}_j^T \mathbf{v}_i = \mathbf{U}(\mathbf{v}_j^T \mathbf{v}_i) = \delta_{ij}$:

$$\mathbf{U}(\mathbf{v}_j^T \mathbf{v}_i) = \mathbf{U}\mathbf{v}_j^T \mathbf{U}\mathbf{v}_i = \begin{bmatrix} U^T v_{1j} \dots \dots U^T v_{nj} \end{bmatrix} \begin{bmatrix} U v_{i1} \\ \dots \\ \dots \\ U v_{in} \end{bmatrix}$$

$$\mathbf{U}(\mathbf{v}_j^T \mathbf{v}_i) = \sum_{k=1}^{n} U^T U v_{kj} v_{ik} = \sum_{k=1}^{n} v_{kj} v_{ik} = \mathbf{v}_j^T \mathbf{v}_i = \delta_{ij} \quad \square$$

One of the unit tests is thus to make sure that orthogonality of the eigenvectors is preserved. We also have a unit test on the eigenvalues of a known symmetric matrix and on `ft_nondiago_largest`. They are in `unittests.cpp`.

[2]The non-diagonal elements are very small, but not zero.

The good thing is that they are tested at any call of the function and that they are successfully passed. They are dependent on the algorithm, but not the matrix we diagonalize. They complete well the invariant test[3]. The tests are ran with the catch function `Catch::Session()`. It computes the tests in `unittests.cpp` and returns 0 if the tests are successfully passed. This result is sent to `main.cpp` with `run_unittest` and used as a prerequisite to compute the algorithm.

2.3 Coding norm and particularities

We took a lot of times to write the program in a developper way. As an instance, our `main` function from the project 1 was huge and there were many different process, or tasks, encapsulated in a single function. It was hard to read, and one needed time to understand what does one part of the code :

```
int main(int argc, char* argv[]){

    // FIRST PART, THE GAUSSIAN ELIMINATION TO OBTAIN OUR VECTOR V

    if(argc!=7){
        std::cout << "ERROR. This program must work with 7 command-line
            arguments for main." << std::endl;
        exit(1);
    }

    int n = atoi(argv[1]);

    double h = 1/(double(n)+1);
    double hSquare = 1/((double)n*n+2*n+1); // so the computer does not
        need to calculate this value a billion times

    double* g = new double[n]; // dynamic array for b tilde
    for(long i = 0; i<n; i++) g[i]=hSquare*f((double(i)+1)*h);

    // WARNING!
    // g[0] in the code does not correspond to g_0 in the reality.
    // it is the first element of the array g, but it contains the value
        of b_1 tilde
    // g[i] in the program in fact corresponds to g_(i+1) in the reality

    double* a = new double[n-1]; // lower diagonal
    double* b = new double[n]; // main diagonal
    double* c = new double[n-1]; // upper diagonal
    for(long i=0; i < n-1; i++) a[i] = atof(argv[2]);
    for(long i=0; i < n; i++) b[i] = atof(argv[3]);
    for(long i=0; i < n-1; i++) c[i] = atof(argv[4]);

    clock_t start, finish;

    // this step consists in making A an upper-triangular matrix by
        forward substitution...
```

[3]Note that - like for the Frobenius norm invariant - we could have compute the orthogonality of the vectors for each call of `jacobi`, but that would be very heavy and the norm test is already a strong mathematical property.

```
start=clock();
for(long i=1; i<n; i++){
    b[i]-=(a[i-1]/b[i-1])*c[i-1];
    g[i]-=(a[i-1]/b[i-1])*g[i-1];
    // no need to change the values of a, even if they change
        mathematically
}

// ...then backward substitution

double* v = new double[n]; // creation of the dynamic array containing
     the v_i, idem, v[i] in the program is v_(i+1) in the reality

v[n-1]=g[n-1]/b[n-1];
for(long i=n-2; i>=0; --i) v[i]=(g[i]-c[i]*v[i+1])/b[i];

// free the memory
delete[] a;
delete[] b;
delete[] c;
finish=clock();

// SECOND PART, EVALUATION OF THE APPROXIMATION

std::ofstream results; // open a stream for the results file
results.open(argv[5]);
std::ofstream data; // open a stream for the data file, a quick
    preview of the results file which is heavy for large n
data.open(argv[6]);

std::string separator="----------------------------------";

results << "Gaussian elimination results for a tridiagonal matrix of
    size n=" << n << ".\nWe have x(0)=0 and x_(n+1)=1." << std::endl
    << std::endl;
results << separator << std::endl;
results << "x value                    num. solution       exact
    solution      relative error" << std::endl;
results << separator << std::endl;

int width=20;
int precision=10;

double maxerror=log10(fabs((v[0]-u(1./((double)n+1))/v[0])));
long mesh=0;

for(long i=0; i<n; i++){

    double xi=(double(i)+1.)/(double(n)+1.);
    double error=log10(fabs(((v[i]-u(xi))/v[i])));
    if(error<maxerror){
        maxerror=error;
        mesh=i;
    }
```

```
        results << std::setprecision(precision) << "x" << i+1;
        results << std::setw(width) << std::setprecision(precision) << xi;
                                    // xi
        results << std::setw(width) << std::setprecision(precision) << v[i
            ];                       // num. solution
        results << std::setw(width) << std::setprecision(precision) << u(
            xi);                      // exact solution
        results << std::setw(width) << std::setprecision(precision) <<
            error << std::endl;          // relative error
    }

    results << std::endl << "Maximum relative error: " << maxerror << "
        for i=" << mesh << "." << std::endl;
    results << "Operation time: " << (double) (finish-start)/((double)
        CLOCKS_PER_SEC) << 's' << std::endl;

    data << "n=" << n << std::endl << std::endl;
    data << "Maximum relative error: " << maxerror << " for i=" << mesh <<
         "." << std::endl;
    data << "Operation time: " << (double) (finish-start)/((double)
        CLOCKS_PER_SEC) << 's' << std::endl;

    results.close();
    data.close();

    delete[] g;
    delete[] v;

    return 0;
}
```

Long and unreadable. Once this constatation made, we tried to apply a norm :

1. A "{" or a "}" must be alone on its line, a "," or a ";" is followed by a space
2. Functions must not be too long and take too much arguments. In this vision, `ft_cosmetics` is a failure (we wanted to use a struct, but we got some trouble)
3. As a consequence big functions must be divided in small functions that compute only one task and that are hopefully regrouped in the same header file. This makes the code more reusable and bugs easier to spot
4. Files must be as compact as they get and well organized so we can jump between them very easily, and they must not have too many functions
5. Only lowercase characters and "_" for the names and camelcase is avoided. We can apply a norm for prefixes : `ft_` for functions, `m_` for methods, etc
6. The declarations must be on top of the function and we avoid multiple declarations and assignations

With this philosophy, everybody can have a clear view very quickly of our algorithm since it is a sequence of explicit blocks. As developers we do will be able to understand it in ten years from now. Here is our `jacobi` function :

```
double* jacobi(mat& A, mat& R, const double epsilon, ofstream& stream)
{

    // declarations
    long const n = A.n_cols;
    long k = 0;
    long l = 1;
    long iterations = 0;
    double largest = ft_nondiago_largest(A, n, k, l);
    double c;
    double s;
    double time;
    // the frobenius norm is an invariant of the transformation
    // we use it as a required test
    double frobenius_norm = norm(A, "fro");
    double* eigenvalues = new double[n];
    clock_t start;
    clock_t finish;


    R = eye(n, n);  // reset the eigenmatrix as a security measure

    start = clock();

    while(fabs(largest) > epsilon && iterations < n*n*n)
    {
        ft_find_trigo(A, c, s, k, l); // finds cos, sin and tan
        ft_rotate(A, n, R, c, s, k, l); // changes the values
        iterations++;
        largest = ft_nondiago_largest(A, n, k, l);
    }


    finish = clock();
    time = (double) (finish-start) / ((double) CLOCKS_PER_SEC);

    // we test the invariant

    ft_invariant(A, frobenius_norm, stream);

    // sorted dynamic array with the eigenvalues

    eigenvalues = ft_diago_to_array(A, n);

    // and write the data into a file

    ft_cosmetics(stream, A, R, n, time, iterations, eigenvalues);

    return (eigenvalues);
}
```

We get the basic steps of the program in a single review of the code.

2.4 Disappointment

If we compare our algorithm with the Armadillo function `eig_sym` we can see that... well we can see that you will definitely never choose our program.

Matrix dimension (n)	Our algorithm		`Arma::eig_sym`	Relative difference
	Op. time (s)	Iterations	Op. time (s)	
20	0.002172	368	0.000043	1.6947
50	0.082457	3541	0.000274	2.4770
100	1.23731	15266	0.001431	2.9363
125	2.93968	24154	0.002005	3.1659
150	6.35067	35292	0.003189	3.2989
200	19.9403	63330	0.004965	3.6037
400	313.938	258845	0.023384	4.1279
500	807.721	407249	0.040245	4.3025

After $n = 125$, the operation times increase significantly. They get more exponential than linear ; we don't even see the Armadillo curve because it is so small compared to Jacobi's one. If we plot the number of iterations, it appears that the operation time is directly linked to the number of iterations. Yet, the iterations-curve is more linear after $n = 100$ than the time-curve, due to the time used to read and write values in a significant matrix. This result is nevertheless quite surprising because we have mathematically no way to tell how many iterations the program will need before we run it, and the curve is very smooth.

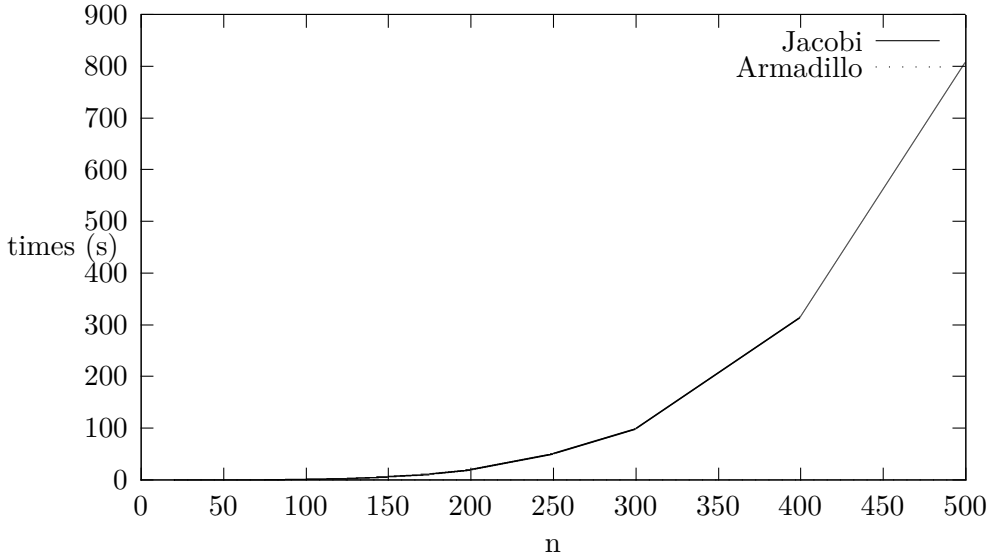

Figure 2.1: Comparison of the operation times of Armadillo and our function.

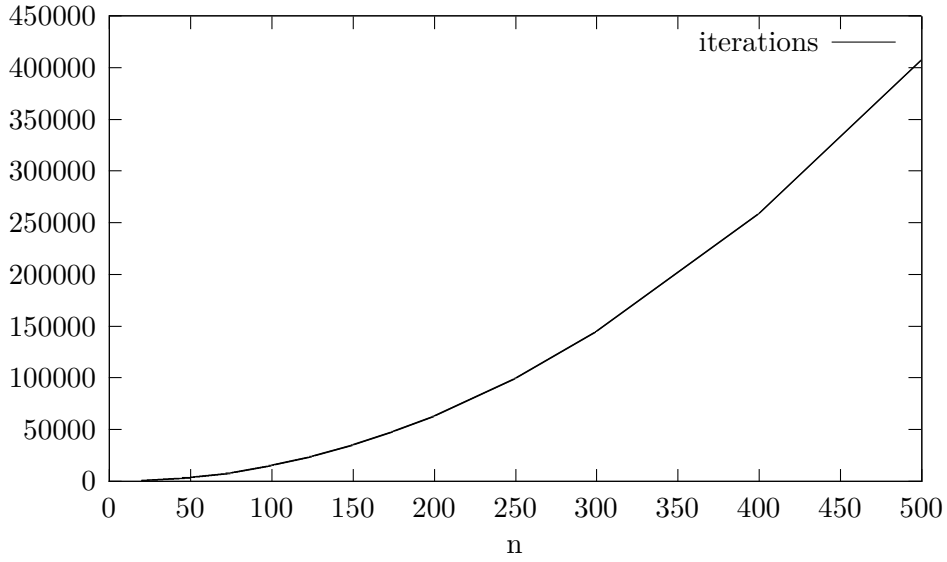

Figure 2.2: Number of iterations needed for a given matrix size n.

Plotting the relative difference of the time-curves of both Armadillo an our algorithm learn us that the behavior is stable regarding a given matrix size : the curve is log-linear which implies that the difference is very stable with respect to the order of magnitude and the size of the matrix. But we clearly understand that Armadillo is so powerful. Our program runs during more than 10mn to compute the eigenvalues of a $500 * 500$ matrix, and Armadillo needs only a fraction of second.

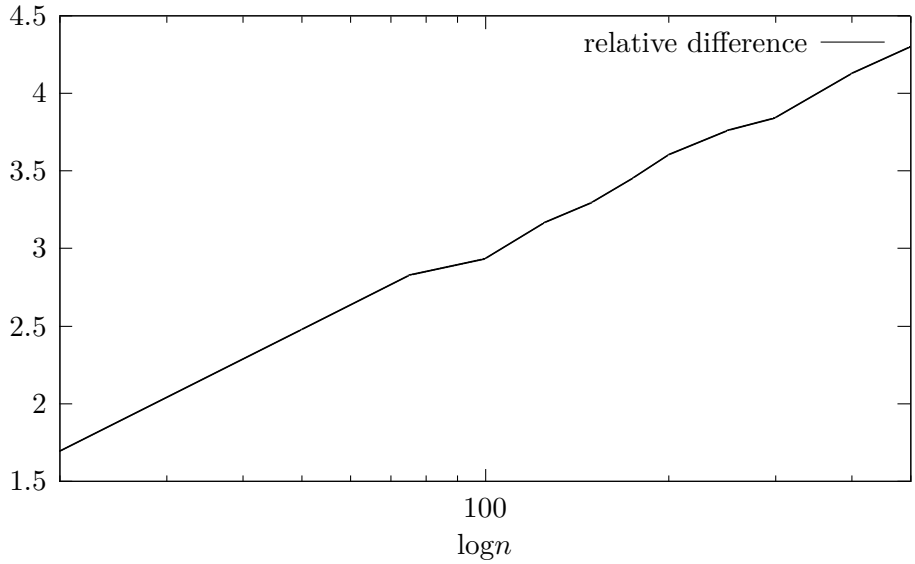

Figure 2.3: Relative difference between Armadillo and our function.

Chapter 3

Results

3.1 Non-interacting case

In the non-interacting case, the values of λ_n should correspond to the energy E_n. We have defined for the equation (2)

$$\lambda = \frac{2m\alpha^2}{\hbar^2}E, \qquad \text{with } \alpha = \left(\frac{\hbar^2}{mk}\right)^{\frac{1}{4}} \quad \text{and } E_n = \hbar\omega(2n+\frac{3}{2}) \quad \text{for } n \in \mathbb{N}.$$

We have then

$$\begin{aligned}
\lambda_n &= \frac{2m\sqrt{\frac{\hbar^2}{mk}}}{\hbar^2}E_n, \qquad \text{with } k = m\omega^2 \\
&= \frac{2m\hbar}{\hbar^2\sqrt{m^2\omega^2}}\left[\hbar\omega\left(2n+\frac{3}{2}\right)\right] \\
&= 2(2n+\frac{3}{2}) \\
&= 4n+3
\end{aligned}$$

$$\lambda_n = 4n + 3 \tag{9}$$

Comparing the numerical results with the analytic results for $N = 200$ meshpoints and $\rho_{max} = 4.2$:

	Analityc results	Numerical results
λ_0	3	2.9999
λ_1	7	7.0002
λ_2	11	11.0255

The choice of $\rho_{max} = 4.2$ is arbitrary : it is impossible to have a perfect ρ_{max} with this program and it can be chosen between 4.1 and 4.3, given a good number of mesh points, typically $N \geq 150$. That being said, those results are very good regarding!

3.2 Interacting case

For this case, we have reused the same algorithm, just changing the value of V to $V = \omega_r^2\rho^2 + 1/\rho$. We compare our result with the analytic solution given in the article by M. Taut in the Physical Review of November 1993. For $\omega = 0.25$ we should find 2×0.6250, according to the Table 1 of the article, for the ground state. If we set $\rho_{max} = 7.85$, we get $\lambda_0 = 1.25$ which is the right result. Taking then $\rho_{max} = 7.85$, we compute the first three eigenvalues for different values of ω :

ω_r	λ	Numerical results	ω_r	λ	Numerical results
$\omega_r = 0.5$	λ_0	2.230	$\omega_r = 3$	λ_0	4.0574
	λ_1	4.1339		λ_1	7.9073
	λ_2	6.0726		λ_2	11.8135
$\omega_r = 1$	λ_0	10.8763	$\omega_r = 5$	λ_0	17.436
	λ_1	22.5764		λ_1	37.0118
	λ_2	34.3796		λ_2	56.7075

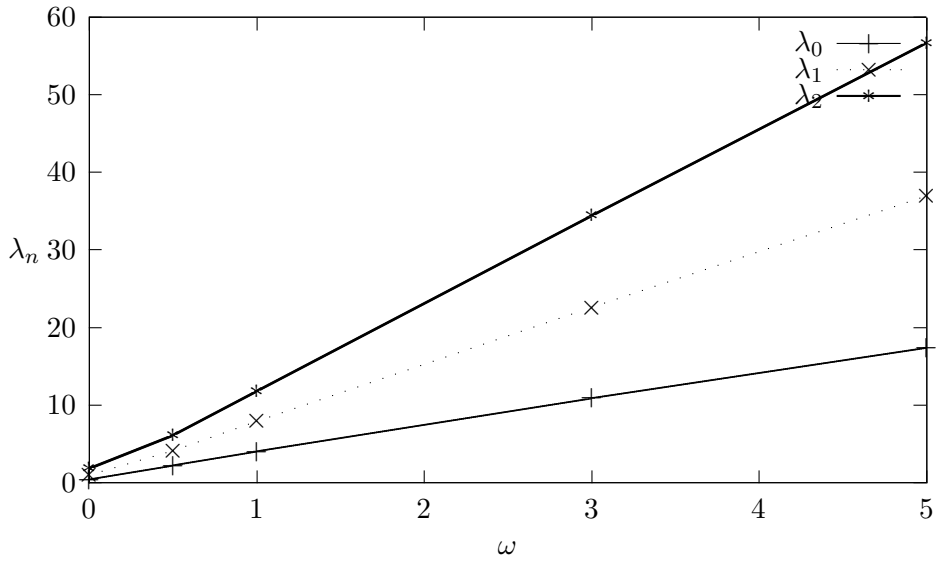

Figure 3.1: Linear evolution of the eigenvalues depending on ω

The result is that each eigenvalues is a linear function of ω such that $\lambda_n = \lambda_n(\omega) = k_n\omega + \gamma$, γ being a small constant. The coefficient k_n is of course different for each eigenvalue λ_n but we observe that is n-dependent as well such that : $k_n = nk$. At the end, we get

$$\lambda_n = n \times k\omega + \gamma \tag{10}$$

Conclusion

This project proves that given any difficult and specific problem such as a quantum physics problem, one does not need to be a specialist to solve the problem, or at least to get an idea on what is going on. Scaling and discretizing transforms the specific problem to a known one. After that we only need to write a general program that will be fully reusable.

This program enabled us to compute very good results for our problem : the eigenvalues we come up with are the good ones and we could find the optimal ρ_{max} for any configuration. The next step is to compute the wave function which is given by the associated eigenvectors.

Bibliography

[1] David J. Griffiths. Introduction to quantum mechanics – second edition. 2016.

[2] M. Taut. Two electrons in an external oscillator potential. *Physical review A*, 48(5), 1993.