FYS 3150 - Project 2
Jacobi method

September 30, 2019

Øyvind Elgvin

Abstract

This study investigates the eigenvalue problems which arise in a typical two point-value boundary buckling beam problem, and the quantum dot harmonic oscillator problem in three dimensions with one and two electrons. Toeplitz matrices are diagonalized and solved with the Jacobi's rotational algorithm, and equations are scaled to fit into a numerical computation model by making variables dimensionless and introducing units which are more convenient. Unit tests are deployed to verify the math and algorithms throughout the project. The results are that the accuracy of the eigenvalues and also time increase with a higher number of integration points, and that Jacobi's algorithm is advisable only for a relatively low number of integration points.

Introduction

The overarching idea in this study is to take real-world problems and transform them into eigenvalue problems, which then is solvable with methods like the Jacobi's rotation algorithm. The fun part is to see how easy it was to take different problems and use the same process for solving them.

For the buckling beam problem, a differential equation manipulates into an eigenvalue problem, with Dirichlet boundary conditions and a tridiagonal Toeplitz matrix, but for the quantum mechanics problem, Schroedinger's equation shapes into an eigenvalue problem with a little manipulation and some defining of variables.

The Jacobi's rotational algorithm diagonalizes a real symmetric matrix which gives eigenvalues and associated eigenvectors.

This report is organized pretty standard with an abstract, introduction, formalism, code implementation, analysis, conclusions, appendix, and bibliography.

Formalism

The first thing to establish is that a unitary transformation also called a similarity transformation, preserves the orthogonality of the obtained eigenvectors, as shown in the appendix. The unitary transformation lets Jacobi's algorithm [1] zero out the non-diagonal

elements starting with the highest valued element to generate a diagonalized matrix, B, by multiplying the original symmetrical tridiagonal matrix, A, with the rotational matrix, Q, in a similarity transformation enough times:

$$\hat{B} = \hat{Q}^T \hat{A} \hat{Q}$$

where for a N = 3 rotation matrix:

$$\hat{Q} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

so the equation actually looks like this:

$$Q_N^T \dots Q_1^T A Q_1 \dots Q_N = D$$

This leads to

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

Theta is then chosen so that all non-diagonal matrix elements, b_{kl}, becomes zero, which results in a matrix, B, with eigenvalues left on the diagonal. The matrix, B, is defined as the variable, A, in the code.

To get the original differential equation into a general eigenvalue problem it is important to scale the equation correct, so:

$$\rho = \frac{x}{L}$$

where the boundaries ar set:

$$\begin{aligned} \rho_{min} &= 0 \\ \rho_{max} &= \text{variable} \end{aligned}$$

In addition is:

$$\begin{aligned} t &= \tan\theta \\ s &= \sin\theta \\ c &= \cos\theta \end{aligned}$$

Which then leads to:

$$t = -\tau \pm \sqrt{1+\tau^2}$$

$$c = \frac{1}{\sqrt{1+t^2}}$$

$$s = tc$$

This method could, In theory, solve a dense symmetrical matrix but for this project, a tridiagonal symmetrical matrix is used, possibly to speed up the diagonalizing.

For Schroedinger's equation the set up is one, and later two, electrons confined in space by four electric fields and interacting with each other only through the repulsive coulomb potential in a harmonic oscillator. Spherical symmetry is assumed for the electrons.

To find the potential the center of mass is neglected and the focus is on the radial part of the Schroedinger's equation, so it looks like this:

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

and with alpha defined as

$$\alpha = \left(\frac{\hbar^2}{mk}\right)^{1/4}$$

the equation turn into

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

where

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

which is an eigenvalue problem of the form

$$\hat{A}\hat{x} = \lambda\hat{x}$$

The beauty of Jacobi's algorithm is that with the scaled equation the equation works as a general eigenvalue solver and it is only necessary to add a potential to the already tridiagonal matrix for the algorithm to find the eigenpairs.

Code, implementation and testing

The complete code is in the repository in GitHub

https://github.com/jkjkjkjkjkjkjkjkjkjk/GitCompPhys/tree/master/Project%202
The main program is named Project_2_atom.cpp

The project is mainly written in c++, and the structure of the project is set up with multiple files, so one file for the main program, one to generate the matrices, one to store the functions, and individual plot files in python to plot the three different problems, 2b, 2d, and 2e.

This section of the report list some of the code to give a sense of how the algorithms are implemented. The algorithm for finding the max off-diagonal sets a k and l for the index with max value:

```
// function for finding the max value of all elements
double maxoffdiag(mat A, int & k, int & l, int N){
    double max = 0.0;
    for (int i = 0; i < N; ++i){
        for (int j = i+1; j < N; ++j){
            if (fabs(A(i,j)) > max){
                max = fabs(A(i,j));
                l = i;
                k = j;
            }
        }
    }
    return max;
}
```

Jacobi's algorithm is implemented pretty straight forward as described above and in the lecture notes :

```
// making the eigenvector matrix R
void jacobi (string filename, mat A, mat R, int n){
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            if (i == j){
                R(i,j) = 1.0;
            }
            else {
                R(i,j) = 0.0;
            }
        }
    }
    // Loop for rotating the matrix
    int k, l;
    double tol = 1e-8;
    int iterations = 0;
    double max_nr_itera = double (n) * double (n) * double (n);
    double max_offdiag = maxoffdiag (A, k, l, n);

    while ( max_offdiag > tol && double (iterations) < max_nr_itera){
        max_offdiag = maxoffdiag (A, k, l, n);
        rotate (A, R, k, l, n);
        iterations++;
    }
    cout << "# iterations for " << filename << " = "<< iterations << endl;

    // makes two files, one with aigenvalues and one with eigenvectors
    vec A_diag = A.diag();
    A_diag.save(filename + "A_eigvalues", arma_ascii);
    R.save(filename + "R_eigenvectors", arma_ascii);
    return;
}
```

And the rotation function is:

```
// Function for the rotation
void rotate (mat & A, mat & 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;
    }

    // replacing the k and l elements in the matrix
    double a_kk, a_ll, a_ik, a_il, r_ik, r_il;
    a_kk = A(k,k);
    a_ll = A(l,l);
    A(k,k) = a_kk*c*c - 2.0*A(k,l)*c*s + a_ll*s*s;
    A(l,l) = a_kk*s*s + 2.0*A(k,l)*c*s + a_ll*c*c;
    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);
        }
        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;
}
```

Here is the implementation of the function that gives the matrix for the two electrons:

```
mat get_ham_2e(int Dim, double omega, double Rmax){

    int i, j, Orbital;
    double Rmin, Step, DiagConst, NondiagConst, OrbitalFactor;
    Rmin = 0.0;
    mat R2(Dim,Dim, fill::zeros);
    mat Hamiltonian = zeros<mat>(Dim,Dim);
    Step = Rmax / Dim;
    DiagConst = 2.0 / (Step*Step);
    NondiagConst = -1.0 / (Step*Step);
    vec rho(Dim);
    vec potential(Dim);
    for (int i = 0; i< Dim; i++){
        rho(i) = Rmin + (i+1)*Step; // boundary condition
        potential(i) = omega*omega*rho(i)*rho(i) + 1/rho(i);
    }
    Hamiltonian(0,0) = DiagConst + potential(0);
    Hamiltonian(0,1) = NondiagConst;
    for (int i = 1; i < Dim-1; i++){
        Hamiltonian(i,i-1) = NondiagConst;
        Hamiltonian(i,i) = DiagConst + potential(i);
        Hamiltonian(i,i+1) = NondiagConst;
    }
    Hamiltonian(Dim-1,Dim-2) = NondiagConst;
    Hamiltonian(Dim-1,Dim-1) = DiagConst + potential(Dim-1);
     return Hamiltonian;
}
```

For the unit tests, armadillo is used to check the eigenvalues, and the maxoffdiag function is tested manually like this:

```
// tests if the maxoffdiag function picks out the highest element
int k, l;
// makes a identity matrix with one element with value 3
mat test1 = test_maxoff_mat(5);
double testen = maxoffdiag(test1, k, l, 5);
    cout << endl
         << "Max off-diagonal test" << endl
         << "if the test prints out 3, it works " << endl
         << "if not, it doesn't" << endl
         << testen << endl;
```

Here is a run through the program to demonstrate the code so that it is possible to validate and verify the results, where $N = 100$, $\rho_{max} = 8.0$, and $omega = 5.0$

```
1x-193-157-183-8:Program from terminal oyvindengebretsen$  ./exe 100

2b results =
Armadillo eigenvalues 2b =
9.67435
38.6881
87.013

Armadillo used 0.00157901 seconds.
# iterations for 2b = 17664
Jacobi used 0.848439 seconds.

Max off-diagonal test
if the test prints out 3, it works
if not, it doesn't
3

RESULTS:2d
Rmin =       0.0000000
Rmax =       8.0000000
Number of steps =  100
Four lowest eigenvalues with armadillo:
      2.9980380
      6.9901830
      10.976028
      14.955558
# iterations for 2d = 16044

RESULTS:2e
Rmin =       0.0000000
Rmax =       8.0000000
Number of steps =  100
Four lowest eigenvalues with armadillo:
      17.396137
      36.823521
      56.246484
      75.572982
omega = 5.0000000
# iterations for 2e = 13467
1x-193-157-183-8:Program from terminal oyvindengebretsen$
```

Analysis of the Results

2b

The three lowest state is plotted below to show the quantum mechanic phenomenon of waves as energy states.

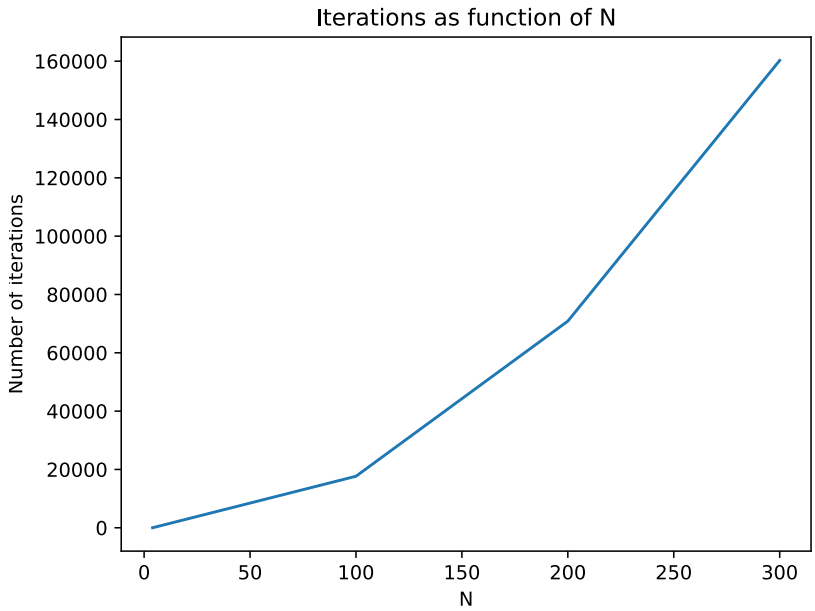

Fig. 1: numbers of iterations vs number of integration points.

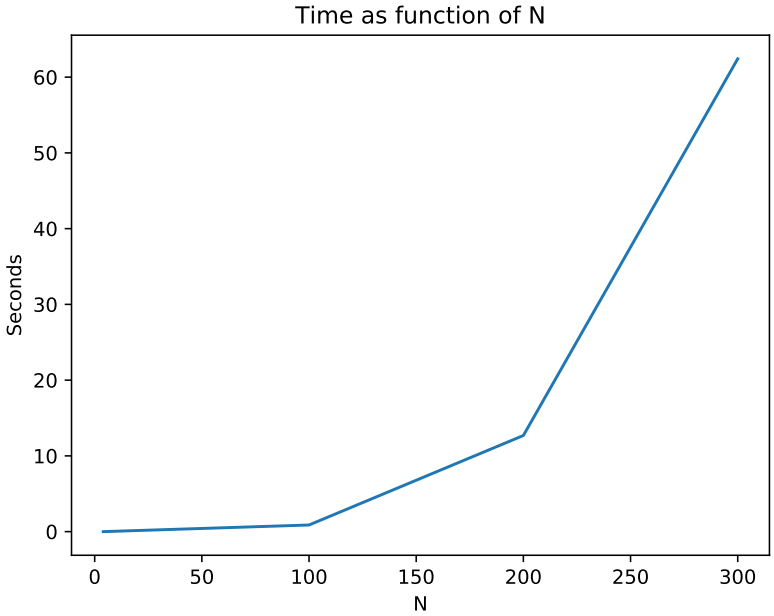

Fig. 2: seconds vs number of integration points.

As presented below in fig 1, and Fig 2, the iterations and time go up with what looks like an exponential curve with the number of integration points.

2c

A unit test is preformed and shows that the Jacobi eigenvalues matches the analytical ones, where GS is short for groups state, as depicted in table 1 in the appendix. A test was also run to check the maxoffdiag function.

2d

With one electron the higher the dimension of the matrix

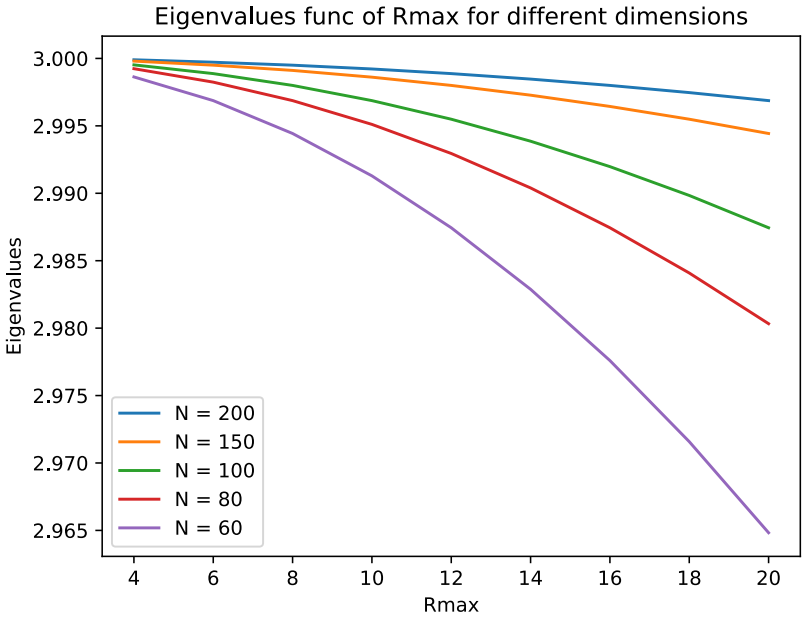

Fig. 2: plot of eigenvalues for different N
Kunne ha plotta error mot

The analytical solution for the ground state for the eigenvalue problem with one electron is 3, so with higher N, the lowest eigenvalue is getting closer to the analytical solution, which the plot also shows. There should potentially be a plot with the eigenvalue error between analytic and approximately vs integration points and rho_max, but that is beyond the time and funding of this project.

Here is the Jacobi ground states for the different N with a static Rmax = 8.

N = 200 the ground state is 2.999505
N = 300 the ground state is 2.999779
N = 400 the ground state is 2.999876
N = 450 the ground state is 2.999902

Around N = 3000 is needed to reach the 4 digits accuracy for all four eigenvalues.

2e

The omega set the width of the energy well so that with a higher omega the well gets narrower.

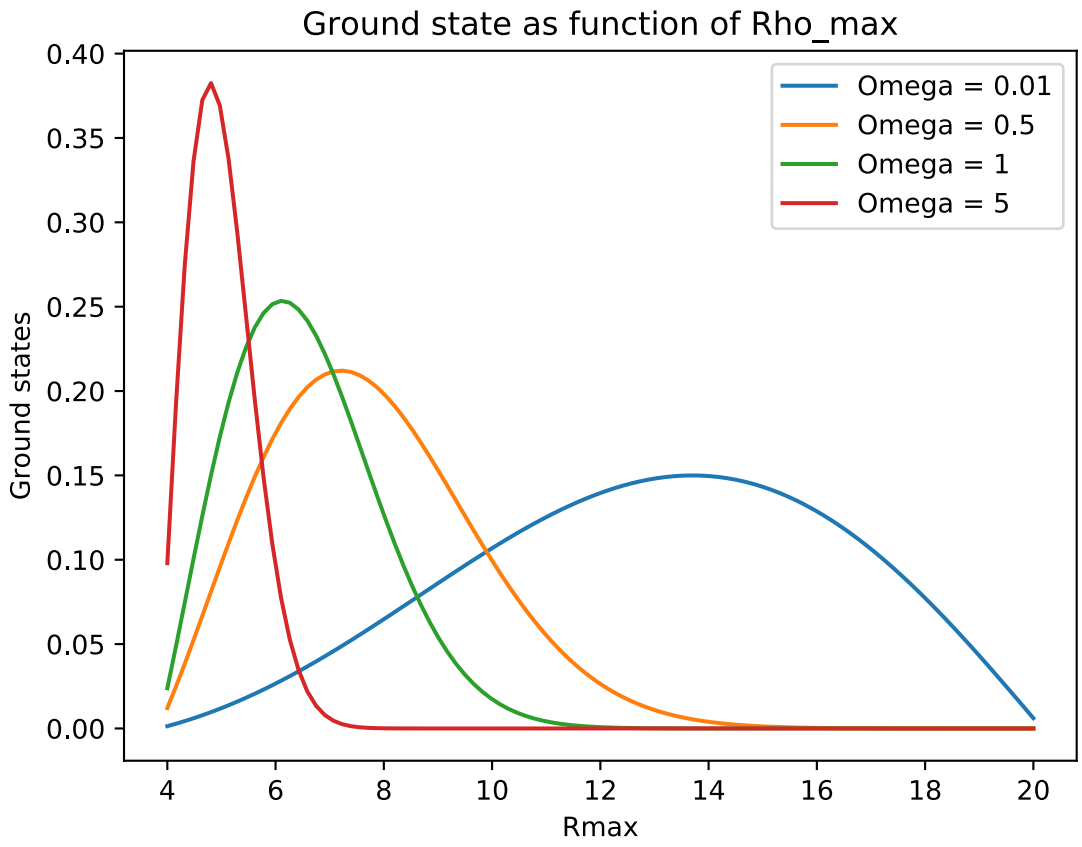

Fig. 3: shows that with a broader energy well the eigenvalues is lower than a more narrower energy well.

The x-axis is probably not correct, as it is just a linspace with the right length without a connection to anything, but what the plot is showing however is that with a lower frequency, a lower omega, the ground state is lower in value as well compared to a higher omega, which gives a narrow energy well and a higher eigenvalue.

I should have compared with the analytical results to Mr Taut, but I didn't understand the relation to his omegas.

Conclusion

2b
For the buckling beam problem both the time it takes to run Jacobi's algorithm, and the number of iterations goes up pretty fast. Jacobi's algorithm finds the analytical eigenvalues but is only advisable for the relatively low number of integration points as the time skyrockets with a significant number of integration points.

2c
Unit tests are important for avoiding errors in the code which can otherwise be hard to find.

2d
For the limited case of only the ground state, the number of integration point to reach the analytical value of eigenvalues within four digits is around 450, with Rmax = 8, but for the second-lowest eigenvalue, the integration points need to be around 3000 to reach a four-digit accuracy for the four lowest eigenvalues.

2e
With two electrons the eigenvalues got higher with a narrower energy well, where a high omega gives a narrow energy well.

Appendix

If

$$w = Uv$$

where

$$\delta = v^T v$$

and

$$w^T = v^T U^T$$

then

$$\begin{aligned} w^T w &= v^T U^T U v \\ &= v^T v \\ &= \delta \end{aligned}$$

for a matrix U which

$$U^T U = I$$

and

$$U^T = U^{-1}$$

and the orthogonal transformation is shown.

N	Armadillo	Jacobi	Iterations	GS Armadillo	GS Jacobi
4	0.00011251	0.000923057	7	6.11146	6.11145618
100	0.00161297	0.87475	17664	9.67435	9.6743541602
200	0.00812958	12.6909	70859	9.77144	9.7714447477
300	0.0122817	62.4196	160258		

Table 1: shows the time for the different algorithms as well as number of integration points.

Bibliography

[1] https://gdz.sub.uni-goettingen.de/id/PPN243919689_0030?tify={%22panX%22:0.509,%22panY%22:0.733,%22view%22:%22info%22,%22zoom%22:0.519}