Fys4150
Project 2

Peter Killingstad and Karl Jacobsen

https://github.com/kaaja/fys4150

September 29, 2017

Abstract

2nd order differential equations leading to symmetric tridiagonal matrices are solved. The method of scaling was applied to the physical equations, and shown to be effective in making different physical problems comparable. Vectorization is shown to speed up simulation speed by a factor of four. The Sturm-Bisection method is shown to be much faster than the slow Jacobi-algorithm. The Sturm-Bisection method is shown to be sensitive to overflow. Testing with catch is implemented.

1 Introduction

2nd order differential equations whose discretization leads to large sparse tridiagonal matrices appears a lot of cases in physics. The matrices are often of such order that it is impossible to solve them on most computers. Being able to solve these physical problems on normal compputers relies on knowledge about special solution methods for these kind of matrices. In this project we learn and implement some of these methods.

We solve sparse tridiagonal linear matrix system resulting from the discretizayion of 2nd order differential equations from quantum mechanics by the method Jacobi and by the the Sturm-Bisection method. The main ideas of both methods are described.

In scientific programming, the simulation times are typically very long (days, weeks). Developing efficient code is central! One method of speeding up the code is the method of vectrorization. In this project we apply this method, and the performance with and without vectorization is compared. Vectorization is shown to give a speed increment of a factor of four.

Efficiency is not always everything. Numerical precision is even more important than efficiency. What is the use of a super fast solver if the result is dead wrong? Knowledge about when to expect issues with numerical precision is very important. Of the two algorithms mentioned above, the Bisection method, which is shown to be much faster than the slow Jacobi-method, gives overflow when the matrices get large. We learned from Barth et al. (1967) [3] about the issue with overflow in the Sturm-Bisection method, and this effect is also illustrated in this report.

Numerical precision problems are one source of errors, but there is one sourse that is, at least in our case, more frequect: human errors. Good testing procedures is necessary for correct results. We apply the testing library catch through this project.

When working with mathematical solutions, the more general the method and solution, the more problems these can be applied to. Scaling is a method that increases the generality of the solutions. We learn about the powers of scalings in this project by using scalings of three different physical problems. These scalings made it easy to implement one common code for all three problems.

2 Theory

2.1 Scaling

In this project we apply the methods of scaling. By typically introducing new non-dimensional variables, as functions of the variables of the original equation, scaling changes the original equations to a form that makes them more general. The new equations are typically less complicated and easier to work with. Also, and not the least, very dissimilar equations representing very different physical problems can after scaling become very similar. So what is this fuzz about? Well, this means that if we can scale a problem properly, and solve it, one has at the same time solved a bunch of other problems! All one have to do when one changes the problem, is to rescale the scaled solution with the relevant variables for the current problem!

In this project we work with three different physical cases, and we scale the equations in all cases. The main advantage we experienced by doing the scalings, is that the number of parameters were drastically reduced. This made the implementation of the solver easier.

A typical good scaling would be a scaling that makes the dimensionless variables' order of magnitude equal to one. Prefferably that dimensionless variabels varies around 0 and 1. In this case, the dimensionless parameters infront of the dimensionless variables represents the strength of each term. This makes it easy to judge the relative importance of the the terms in the equations.

2.2 Vectorization

A very easy and powerfull way to increase the computational efficiency is to apply so-called vectorization. The standard method of compiling a program will use so called scalar operations of computing data. The scalar operations handles typically one iteration at the time. Vectorization however, lets us use multiple operations simultanuously. When operating on arrays in a for loop, a vectorized program would then handle multiple operations at one time.

If we want to vectorize our program, we can give optimalization flags to let the compiler know that we want to vectorize our program, i.e:

```
c++ -O3 project.x project.cpp
```

or

```
c++ -Ofast project.x project.cpp
```

In some cases with matrices and arrays inside loops applying vectorization can be problematic. So-called read after write statements are examples of cases where vectorization can be problematic. Example: Say one array element depends on the previous array element, then a line that directly relates these two arrays will fail when applying vecorization. However, there is a remedy: temporary variables! We applied temporary variables in the Jacobi-algorithm so that we could us vectorization.

2.3 Orthogonal tranformation properties

In at least one of the algorithms that will be used in this report, orthogonal transformations will be applied. A central property of orthogonal transformations is that they preserve the dot product and orthogonality. This will now be shown.

We have an orthogonal tranformation

$$A = Q^T U Q, \tag{1}$$

$$Q^T Q = I. \tag{2}$$

The orthogonal tranformation of our orthogonal unit vectors is

$$\mathbf{w}_i = \mathbf{U}\mathbf{v}_i,$$

Now lets calculate the dot product of the orthogonally transformed unit vectors.

$$\begin{aligned}
\mathbf{w}_j^T\mathbf{w}_j &= (\mathbf{U}\mathbf{v}_j)^T\mathbf{U}\mathbf{v}_i && \text{(3a)}\\
&= \mathbf{v}_j^T\mathbf{U}^T\mathbf{U}\mathbf{v}_i && \text{(3b)}\\
&= (\mathbf{v}_j)^T\mathbf{v}_i && \text{(3c)}\\
&= \delta_{ij}, && \text{(3d)}
\end{aligned}$$

so the dot-product of the transformed orthogonal unit vectors is unchanged by the transformation.

2.4 Jacobi's algorithm

With this method, orthogonal transformations, represented by standard rotational matrices, are applied. Applying these transformations we want to make the original matrix simimlar to a diagonal matrix. From standard linear algebra we know that in such cases, the diagonal elements on the new matrix equals the eigenvalues of the original matrix. But this case is in most cases not as easy as the standard linear algrabra case. One single orthogonal transformation will for larger matrices than $2x2$ not be sufficient to create a similar matrix that is on diagonal form. One needs to do many transformations to get the wanted diagonal similar matrix.

However, with the property above, the preservation of inner product and orthogonality of orthogonal transformations, one can show that the eigenvalues are preserved irrespective of the number of transformations. This validates Jacobi's method, which applies a series of transformations to create a diagonal matrix similar to the original matrix.

The way the transformations, or more precisely, the rotations, in Jacobi's method is decided, is to find a rotation angle that makes the largest non-diagonal element of the transformed matrix equal to zero. When this angle is found, one applies this rotation on the original matrix, updating the original matrix. Now the same procedure is repeated, by zeroing out the largest non-diagonal term of the updated matrix.

At first sight this sounds all good, and one might think that one only needs to do as many rotations as the number of off-diagonal elements in the original matrix. Wrong! A new rotation affects the non-diagonal element that was zeroed out in the previuous rotation!

The above is problematic, but the comfort is that it can be shown that the non-diagonals gets smaller and smaller with this method, so that the method converges! But the convergence will typically be slow. According to Hjorth-Jensen [1] p. 217 the algorithm typically needs $12n^3 - 20n^3$ FLOPS.

The algorithm is described in Hjorth-Jensen [1] p. 217, and our implementation is found in the file 'jacobi.cpp'

Finally it should be mentioned that there are versions of Jacobi's method for eigenvalues that are more efficient than the method described above. One more efficient method is the cyclic Jacobi method, which saves a lot of FLOPS by avoiding searching for the max non-diagonal elements. Instead of searching for the max non-diagonal elements, which by itself is $\mathcal{O}(N^2)$ FLOPS, the cyclic method just runs through row for row and zeroes one non-diagonal at the time.

2.5 The Sturm Bisection method

This is a method for computing the eigenvalues of a symmetric tridiagonal matrices. The eigenvalues are given by the solutions to the characteristic polynomial, which is the determinant of $A - \lambda I$. This is normally inefficient to compute, but this is not necessarely the casefor symmetric trdiagonal matrices.

The characteristic polynomial can be expressed by use of a Sturm sequence. Sloppily put a Sturm sequence is a recursive relation, where each element is represented by a polynomial that depends on the previuous two elements (polynomials). The last term of the Sturm sequence for a matrix is the characteristic polynomial. Sturm sequences

comes handy for us when they are combined with Sturm's theorem.

Sturms theorem says that the number of sign changes in the Stuerm sequence, when comparing all elements, for a given guess on an eigenvalue, equals the number of eigenvalues lower than the guessed eigenvalue. But how do we know where to start look for eigenvalues? This is where Mr. Gersgorin comes to help.

Gershgorin's theorem gives the minimum and maximum eigenvalue of a matrix. A relation between the diagonal's and non-diagonals in each row of the matrix is calculated, giving the so-called Gershgorin's disks, and the theorem says that the eigenvalues are limited by the minimum and maximum of these disks.

So now with Gersghorin's and Sturm's help, we can start looking for the eigenvalues. From Gershgorin we get the max and min eigenvalue. From the Sturm sequence and Sturm's theorem we get the number of eigenvalues lower than a given eigenvalue guess. If we now insert the maximum eigenvalue into the function for Sturms theorem, we will get the total number of eigenvalues. By reducing the eigenvalue guess gradely, the number of eigenvalues will be reduced. We can use the bisection method to identify the value of lambda guess that reduces the eigenvalue number, that is the number of eigenvalues lower than the giessed value, by exatly one compared to the maximum eigenvalue. From this we have gotten the range where the largest eigenvalue is located! Now we do exactly the same for the next largest eigenvalue, and keep going for all eigenvalues, giving us all eigenvalue bounaries.

Having all the eigenvalue boundaries, we can now use the bisection method for each domain in combination with the last element of the Sturm-sequence, remembering that the last term of the Sturm-sequence is the characteristic polynomial. The result should then give...All the eigenvalues!//
The above described algorithm is implemented in the c++ program, 'eigenvalueBisection.cpp' in the following way

- 'gorshgorin': Calculates the min and max eigenvalue from function.
- 'sturmSeq': Calculate the Sturm sequence.
- 'numLambdas': Finds the number of eigenvalues lower than a given guess.
- 'lamRnage': Identifies the domains of all eigenvalues.
- 'Bisection': Finds the roots given the eigenvalue domains.

3 Results

3.1 Algorithm efficiency

The two plots below shows the times used by the different algorithms for different sizes of N. Note that the plot to the right applies $\log_2$.

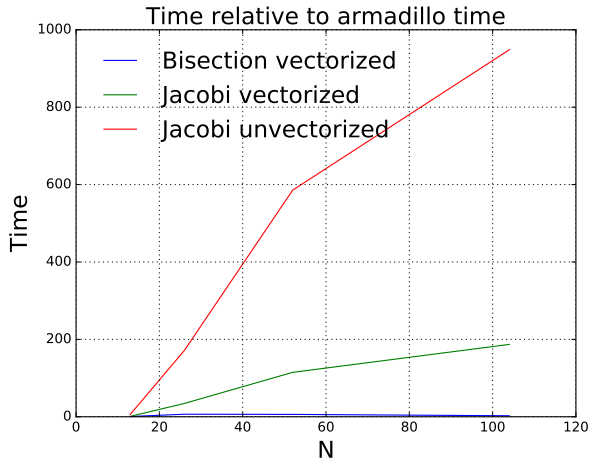

Figure 1: Algorithm times divided by Armadillo time

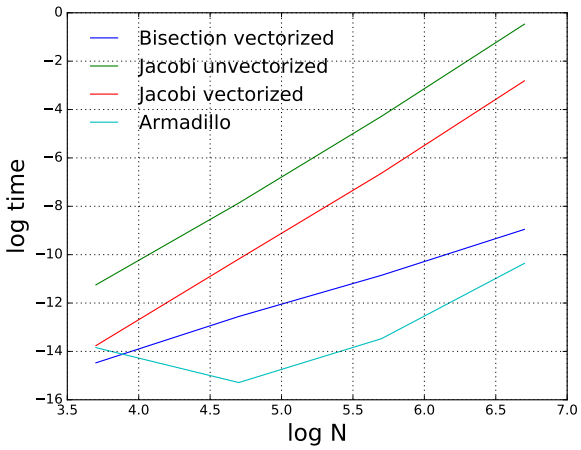

Figure 2: log2 Algorithm times

From the figure to the left we see the power of vectorization straight away. For the case of $N = 100$, the unvectorized solution takes more than four times as much time than the vecotorized solution! From the plot to the right we see that the relative difference between vectorized and unvectorized is constant, so the effect of vectorization is a 'level effect'.

Also we see that the Sturm-bisection method is comparable to Armadillo's standard method for solving eiganvlues when it comes to computational time.

From the figure to the right we see that the order for Jacobi when it comes to time, looks like $\mathcal{O}(N^4)$. This was not expected! Assuming the running time is proportional to the FLOPS, based on Hjorth-Jensen [1] p. 217, which says the solution typically needs $12n^3 - 20n^3$ FLOPS, we expected the slope of Jacobi in the right figure above to be 3. We do not know the reason of this disperancy.

The right figure also suggests that the order of the Sturm-Bisection method is N, which is drastically lower compared to the Jacobi algorithm!

3.2 Sturm bisection

The below figures displays some results from the Sturm-Bisection method for the single electron case, where we know the three smallest eigenvalues.

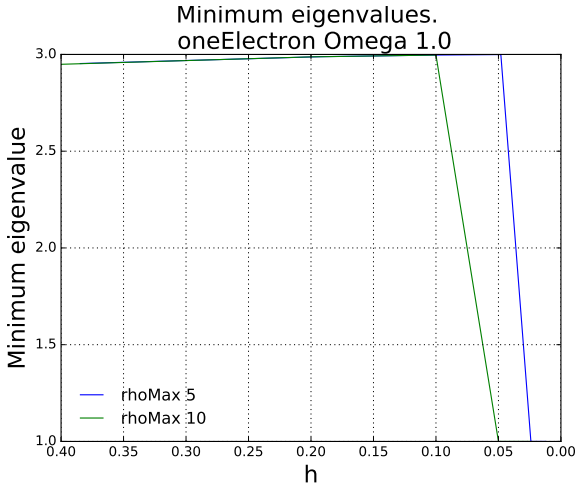

Figure 3: Minimum eigenvalue. One electron Sturm Bisection.

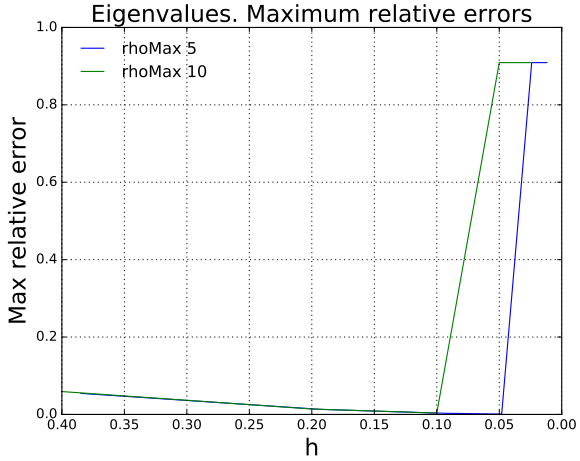

Figure 4: Maximum relative error of three smallest eigenvalues. One electron. Sturm Bisection

The plots above show that the Sturm-Bisection first converges to the correct eigenvalues, but then suddenly they shift to something wrong. Accordring to Barth et al. [3] this is due to overflow in the standard Sturm-Bisection method. They suggest a revised method as remedy. We have tried implementing this revised method, but sadly without success.

3.3 Comparison with theory

3.3.1 ω_r and ρ_{max}

We start our comparison with theory by analyzing the effect of N and ρ_{max} on the solutions. Before looking at some results, a few words about how the values of N, ρ_{max} and, implicitely, h are set.

We adjusted the starting N for each ρ_{max}-simulation such that h in the different ρ_{max}-simulations would be the same. Doing this, we are able to analyse the effect of ρ_{max} at a given mesh refinement, h representing the mesh-refinement.

In the c++ program, for a given ρ_{max}, N is not increased further when the minimum eigenvalue is not changed beyond a given tolerance compared to the results from the previuous N. So this defines our convergence for a given N.

It is still possible that even if the eigenvalue has converged over N, passing the test discussed in the previuous paragraph, that the eigenvalue change when ρ_{max} is changed. To include the possibility of this, we have a for the convergence of the eigenvalues for different ρ_{max} in python where we run everything from. This tests checks whether the eigenvalue for the two last ρ_{max}'s has converged for a given tolerance. If these eigenvalues has converged, and of course if the eigenvalues have converged over N also, the simulations are stopped.

The above four figures shows the results for four different ω's in the two electron repulsion case.

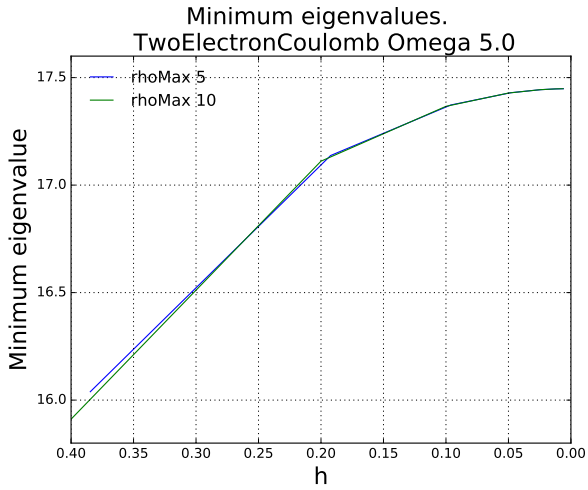

Figure 5: Comparison rhoMax. Minimum eigenvalue. Coulomb

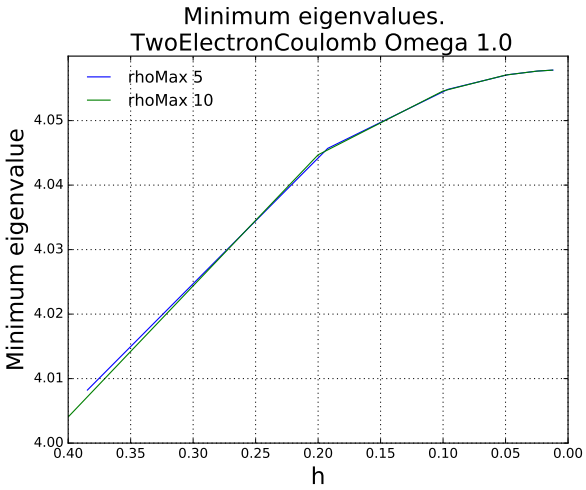

Figure 6: Comparison rhoMax. Minimum eigenvalue. Coulomb

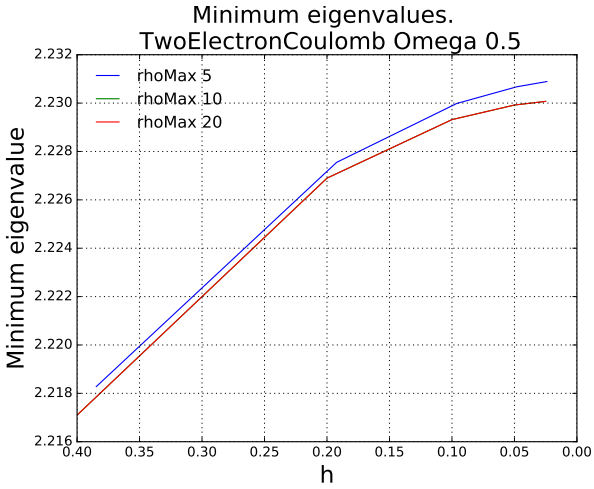

Figure 7: Comparison rhoMax. Minimum eigenvalue. Coulomb

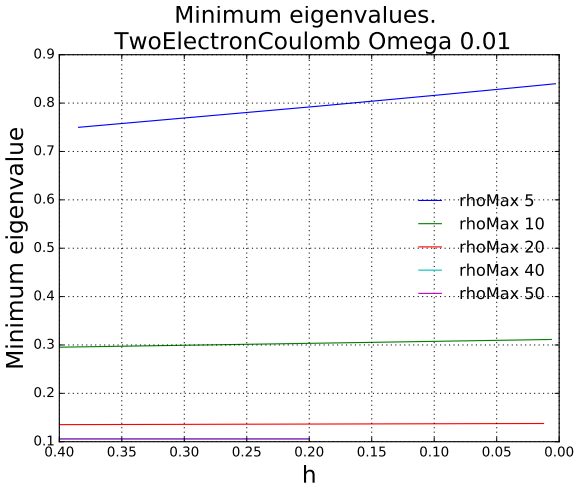

Figure 8: Comparison rhoMax. Minimum eigenvalue. Coulomb

The above figures shows that larger ρ_{max} is needed for convergence when ω gets small. Looking at the scaling, we have $\rho = \frac{r}{\alpha}$. Given r is a destance, α is the characteristic length of the system. (In mechanics we think we would in this case have scaled such that α should make sure that the order of magnitute of ρ is 1, but this does not seem to be the goal and case in this project). We also have

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

giving a relation between α and ω_r. So when ω_r is changed, α is changed, meaning that the characteristic length of the system is changed. This changes ρ, and explains why ρ_{max} depends on ω_r in the simulations. However, we do not see how this relate to convergence of the eigenvalues.

3.3.2 Single electron

The below figure shows the maximum relative error of the three lowest eigenvalues.

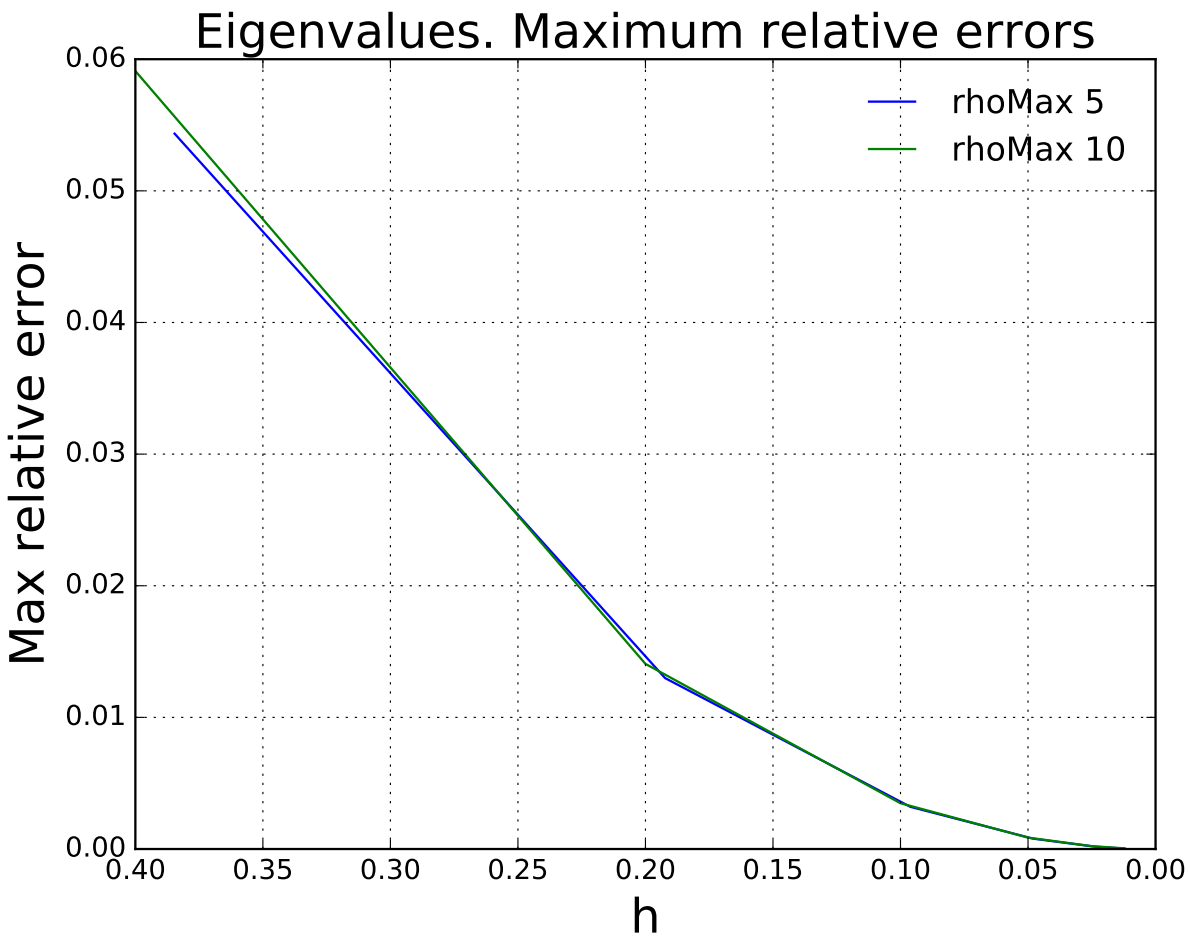

Figure 9: Maximum relative error of the three smallest eigenvalues, one electron.

The figure above shows that the anlytical solution is reached for $\rho_{max} = 10$.

3.3.3 Two electrons and no repulsion

The figures shows the results for the one electron case and the two electron case without repulsion.

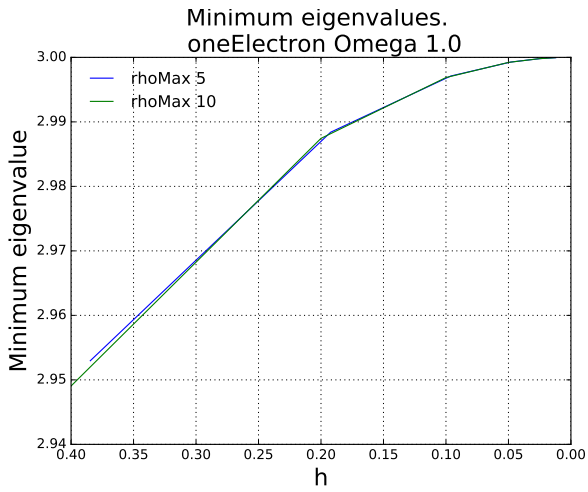

Figure 10: Minimum eigenvalue. 1 electron.

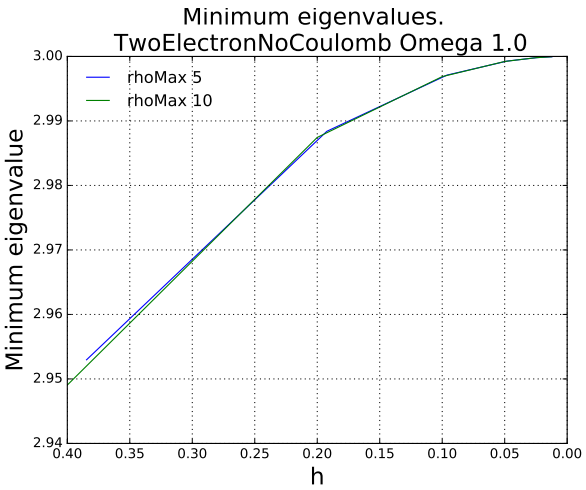

Figure 11: Minimum eigenvalue. 2 electrons no Coulomb.

The results for the one electron case and the two electron without repulsion case, displayed in the figures above, was as expected. When the frequency is the same in the two cases, the equations suggests that the solutions should equal, which they do.

3.3.4 Two electron and repulsion

For the case with $\omega_r = 0.25$ we can compare our results with the analytical results from Taut (1993) [4]. The figure below shows the relative error of the numerical simulations when Tau's result is used as exact results. We note that we had to take into account the difference in scaling by Taut and us, there being a factor of 2 in difference between our smallest eigenvalue and Taut's energy.

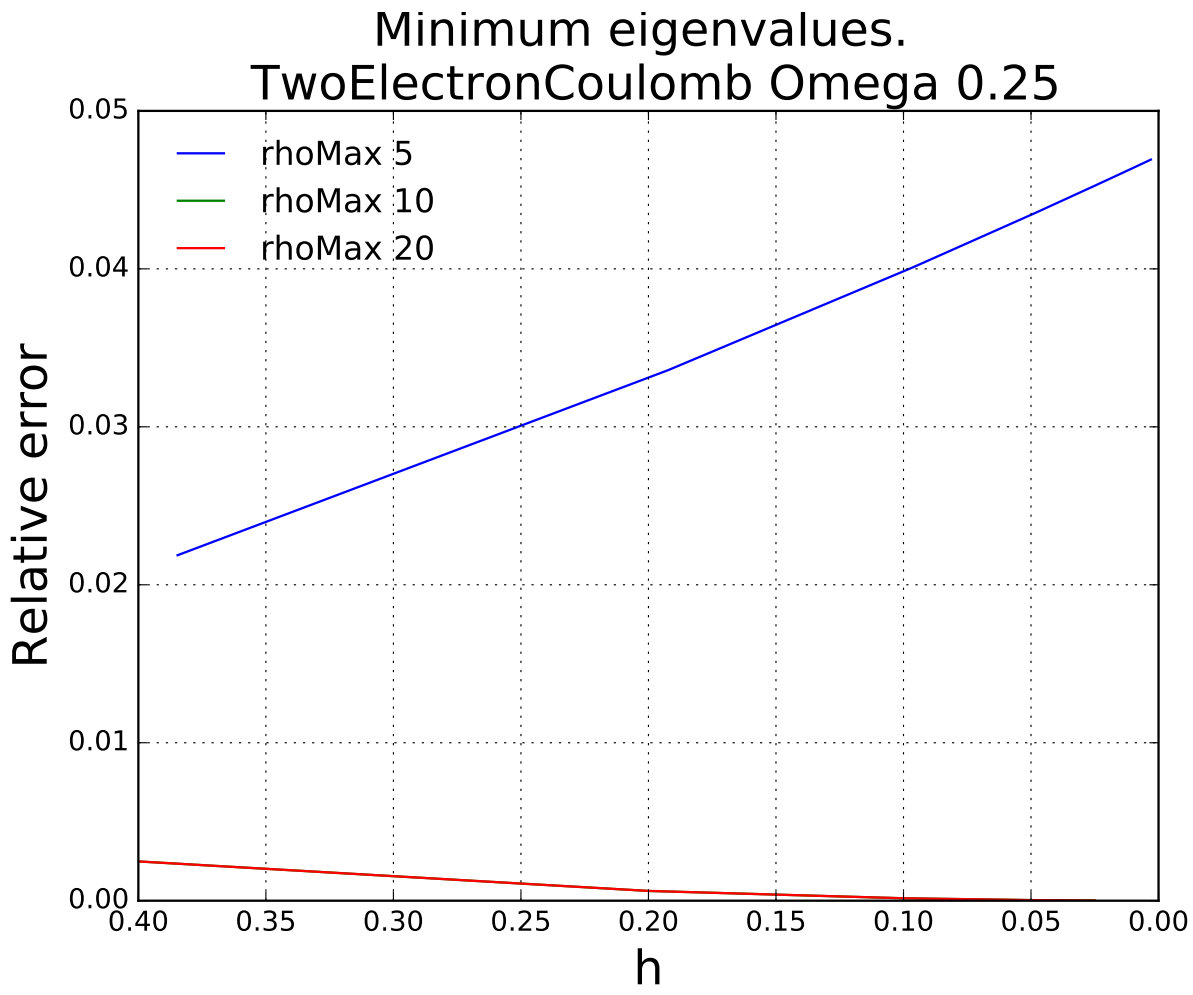

Figure 12: Relative error smallest eigenvalue.

The figure above shows that we reach Taut's result for $\rho_{max} = 20$. Actiually we can say that the result was reached for $_{max} = 10$, since the result is equal, whithin tolerance, for these two ρ_{max}'s.

The below figures shows the results for the convergent solutions for the two electron cases.

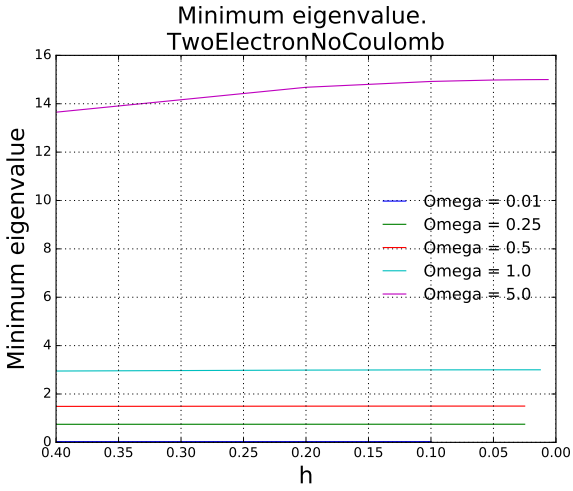

Figure 13: Minimum eigenvalue. 2 electron. No Coulomb.

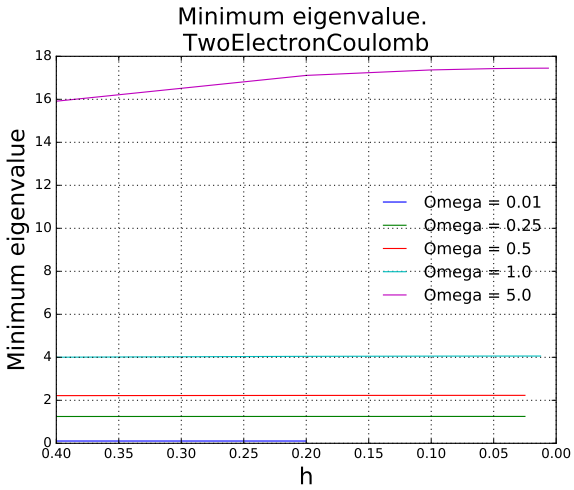

Figure 14: Minimum eigenvalue. 2 electrons Coulomb.

The figures above shows that the eigenvalues increases with ω_r and with Coulomb-repulsion.

3.4 Tests

We applied catch for all tests, and all tests passed.

For the Jacobi-algorithm we tested that the algorithm found the maximum non-diagonal, and that it found the eigenvalues of a $3x3$ matrix. In the last case, we used Armadillo to compute the 'exact' eigenvalues.

For the Sturm-bisection algorithm, all functions were tested.

4 Conclusions

In this project we have solved three different physical problems resulting in 2nd order differential equations that all resultet in a symmetric tridiagonal matrix.

The method of scaling was applied to the physical equations, making the equations very similar. This made it easier to compare the results of the different physical problems.

Vectorization was used, and we found that the simulation time without vectorization was four times larger than with vectorization.

We uesd two different algorithms for solving the linear system: the Jacobi method and the Sturm-Bisection method. The last method was drastically faster than the Jacobi method, which was very slow. However, the last method was shown to be prone du overflow issues when the matrix got big.

Finally, we implemented testing for the first time in c++ by catch. When we got the hang of catch, this was very effective. Testing every new function revealed the error's much, much faster compared to doing a lot of changes before running without any testing.

5 Feedback

5.1 Project 1

This project has been extremely educational. We learned about about c++, especially pointers and dynamic memory allocoation. Also which for us was a well forgotten subject, we learned about dangerous of numerical round-off errors.

We feel the size of the project is large, much larger than typical assignments in other courses. However, the quality and quantity of the teaching without a doubt made the workload managable. The detailed lectures, combined with the fast and good respones on Piazze helped a lot!

We think the project could have gone even smoother, if we on the 2nd lab-session had learned basic branching in Github. We used a considerable amount of time finding out of this.

All in all, two thumbs up!

5.2 Project 2

- catch: We ended up using a lot of time making this work properly. Still we have some problems with catch and Qt. We think we might had benefited from a demonstration at the lab.
- After reading the lecture notes, it is still not fully clear for us how the Lanczos algorithm gives eigenvalues.
- We were not able to understand the Sturm-Bisection algorithm from Barth et al.'s [3] paper on the revised Sturm-Bisection.

- Aparts from the small details above, we are very happy about this project. It feels like a very good use of invested time!

6 Bibliography

[1] Hjorth-Jensen, M.(2015) *Computational physics. Lectures fall 2015.* `https://github.com/CompPhysics/ComputationalPhysics/tree/master/doc/Lectures`

[2] Kiusalaas, J.(2013) *Numerical Methods in Engineering with Python 3. 3rd edition.*

[3] Barth, Martin, Wilkinson (1967) Calculation of eigenvalues of a symmetric tridiagonal matrix by the method of bisection *Numeriche mathematik 9, 386 - 393 (1967)*

[4] Taut, M. (1993) Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem *Phys. Rev. A 48, 3561 (1993).*