Fys4150
Project 2

Peter Killingstad and Karl Jacobsen

`https://github.com/kaaja/fys4150`

October 20, 2017

Note to instructurs about Github repository

If the above Github-link does not work, it is eighter because you have not yet accepted our invite to the repository, or you have not yet provided us with an e-mail adress available at Github so that we can invite you. If the latter applies to you, please send us an e-mail with an e-mailadress available in Github or your Github username so that we can send you an invite. Our e-mailadresses: peter.killingstad@hotmail.com, karljaco@gmail.com.

Abstract

Eigenvalue problems with 2nd order differential equations leading to symmetric tridiagonal matrices are solved. Eigenvalues are computed with Jacobi's method, the Sturm-sequence method, and Lanczos' method. 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 and Lanczos' method are both shown to be much faster than the slow Jacobi-algorithm. The Sturm-Bisection method shown to be sensitive to overflow. Lanczos method needed to be implemented with a twist to work properly. Testing with 'catch' is implemented.

1 Introduction

Physical problems are often eigenvalue problems. Often the discretization of these eigenvalue problems leads to large sparse tridiagonal matrices. The matrices are often of such order that it is on most computers impossible to solve the eigenvalue problems whithin reasonable time limits applying the most basic method. 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.

Full understanding of how an eigenvalue algorithm works is important. Without such understanding it is hard to understand how other eigenvalue solvers work and what to expect for results. In this project we learn and implement fully Jacobi's algorithm for finding Eigenvalues of symmetric matrices.

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 vecotization, and the performance with and without vectorization is compared. Vectorization is shown to give a speed increment of a factor of four.

Also algorithms can speed up simulations drastically. We discuss and implement two such algorithms that are faster than the Jacobi-algorithm: The Sturm-Bisection method and Lanczos method.

Efficiency is not everything. Numerical precision is even more important than efficiency. What is the use of a super fast solver if the results are 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) [1] 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. By scalings three different physical problems, we learn about the powers of scalings in this project . These scalings made it easy to implement one common code for all three problems, and it also allowed us to solve two different problems with only one simulation!

2 Theory

2.1 Scaling

In this project we apply the method of scaling. By 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. Compared to the original equations, the scaled 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. 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 solve for the scales. So one scaled solution is potentially the solution to many different physical problems! An important assumption here is that the problem must be properly scaled. If the problem is badly scaled, it will not be possible to rescale it back into somthing physical meaningful.

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. The differences between seemingly different problems is in the scaled equations represented typically by one or two parameters. This makes comparison 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 both the Jacobi algorithm and in Lanczos' algorithm orthogonal transformations are 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. However, obtaining the diagonal matrix is in most physics scenarieros 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. But will the eigenvalues ble preserved after multiple transofrmations?

With the results from the previuos section, the answer is yes! 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! New rotations 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 [2] p. 217 the algorithm typically needs $12n^3 - 20n^3$ FLOPS.

The algorithm is described in Hjorth-Jensen [2] p. 217, and our implementation is found in the file 'jacobi.cpp'

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 determinanant is for large matrices 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. In its own the Sturm sequence is not useful for us, since we don't want to compute the solution to the characteristic polynomial, the last term in the Sturm sequence, directly. However, the 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 Sturm 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. The theorem states that the eigenvalues are limited by the minimum and maximum of the Gershgorin 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 the eigenvalue 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.

2.6 Lanczos' method

Lanczos's method applies to symmetrical matrices. The method tridagonalizes a symmetric matrix. But triagonalization of a symmetric matrix is among practioners preffered done with Householder's method. The thing with

Lanczos's method is that the method can produce tridiagonal matrices of lower dimensions than the original matrix. So why would we do that?

The magic thing is that the extremal eigenvalues of the tridiagonal matrices created by Lanczos' method converges to the extremal eigenvalues of the original matrix! And what is so nice about that?

Say the original matrix is huge. Then if one is interested in the extremal eigenvalues of this big matrix, one can by use of Lanczos's method create a much smaller tridiagonal matrix and then easily compute the eigenvalues of this smaller matrix at a low computational cost. The extremal eigenvalues of this small(er) tridiagonal matrix would then approximate the extremal eigenvalues of the big matrix.

There is one potential issue with Lanczos' algorithm. The algorithm requires calculation of Aq. If A is large, which it probably is, since one wants to compute its eigenvalues by an alternative method, evaluating Aq can be very expensive. So for Lanczos' algorithm to be efficient, one needs an efficient way of calculating Aq.

In our case, where A is tridiagonal, we can easily calculate Aq by $Aq(k) = A(k, k-1)q(k-1) + A(k, k)q(k) + A(k, k+1)q(k+1)$, with slightly modified formulae for the first and last row. Assuming the equation is the same for the first and last row, for simplicity, the above formula gives three multiplications and two summations for the N rows of A, resulting in $5N$ FLOPS. In comparison, computing Aq would have required N multiplications with $N-1$ additions for N rows, resulting in $N^2(N-1)$ FLOPS, which is considerable more then the $5N$ FLOPS required with the alternative method.

A potential problem with Lanczos' method, is that when the number of itertaions grow, the orthogonality of the new Lanzcos' is lost. This potentially destroys the eigenvalue results. A remedy for this is to use the Gram-Schmidt process to orthonormalize the new Lanczos vectors. This is done in our implementation.

The formal derivation and algorithm for Lanczos' method is described in p. 226-227 in Hjort-Jensen [2].

3 Results

3.1 Scaling

In the one electron case, when having separated variables and set $l = 0$, we have the following unscaled equation

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dr^2} + (1/2)kr^2\right)u(r) = Eu(r). \tag{4}$$

Introducing the non-dimensional variable ρ, $\rho = r/\alpha$, where α is a characteristic lenght, the equation becomes

$$\left(-\frac{d^2}{d\rho^2} + \frac{mk}{\hbar^2}\alpha^4\rho^2\right)u(\rho) = \frac{2m\alpha^2}{\hbar^2}Eu(\rho). \tag{5}$$

The scaling is done quoting Horth-Jensen [3] p. 2 :

" The constant α can now be fixed so that

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

or

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

Defining

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

we can rewrite Schroedinger's equation as

$$-\frac{d^2}{d\rho^2}u(\rho) + \rho^2 u(\rho) = \lambda u(\rho). \tag{6}$$

"

Not having experience with quentum physics, we are not sure of the interpretation of the scaling choise made above. The choise gives an expression for α, and since α is the characteristic length of the system, if the coise is good, the α should give a meaningful characteristic length. However, the radius r should go to infinity, so it is not so easy to intpret what a reasonable α whould be. Regardless of this discussion, the soultion to the above equation holds for all physical problems corresponding to the chosen scaling. Example: We can find the solution for difference m's if we can change k to keep α unchanged. So for the given α, we have the solution to all m and k combinations.

Now lets move on to the two electron cases. After variable separation we have

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} + \frac{1}{4}kr^2 + \frac{\beta e^2}{r}\right)\psi(r) = E_r\psi(r), \tag{7}$$

where $\frac{\beta e^2}{r}$ is the Coloumb-repulsion.

Again the non-dimensionalization $\rho = r/\alpha$ is introudced, and insertion of this into the above equation gives

$$\left(-\frac{d^2}{d\rho^2} + \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4\rho^2 + \frac{m\alpha\beta e^2}{\rho\hbar^2}\right)\psi(\rho) = \frac{m\alpha^2}{\hbar^2}E_r\psi(\rho). \tag{8}$$

Scaling is done by quoting Hjorth-Jensen [3] p. 6

" We want to manipulate this equation further to make it as similar to that in (a) as possible. We define a new 'frequency'

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

and fix the constant α by requiring

$$\frac{m\alpha\beta e^2}{\hbar^2} = 1$$

or

$$\alpha = \frac{\hbar^2}{m\beta e^2}.$$

Defining

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

"

Schroedinger's equation can be rewritten to

$$\left(-\frac{d^2}{d\rho^2} + \omega_r^2\rho^2 + \frac{1}{\rho}\right)\psi(\rho) = \lambda\psi(\rho). \tag{9}$$

A few observations. First we note that a different scaling is chosen in the two-electron cases than in the one electron case. This can be seen by the choise of α. α differs between the electron cases. Secondly we note that (6) and (9) look very similar. Removing repulsion and setting the new frequency $\omega_r = 1$ makes the equations identical!

This means that the solution to the *scaled* one electron case equals to the solution of the *scaled* version of the two electron case without repulsion and $\omega_r = 1$. This DOES NOT mean that the unscaled solutions are the same! To obtain the unscaled soulutions, one would have to factor back the scalings, and these scalings are, as mentioned, not identical in the two cases. So for the two electron case one would have to use the α for that case, and for the one electron case one would have to use the α for that case. However, factoring back scalings is an easy job. So by solving one scaled equation for two electrons without repulsion and $\omega_r = 1$, we not only get the solution for all physical problems in the two-electron non-repulsion $\omega_r = 1$ scenarios that satisfy the ccaling, we also the the solution to all the one-electron cases!

One important note, which has not been stated directly above, is that the non-dimensionalization reduces the number of parameters in the equations we work with. This makes life easier.

3.2 Algorithm efficiency

Before commenting on efficiency results, a few words about the Jacobi-implementation. The Jacobi algorithm is implemented in 'jacobi.cpp'. 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. The tests are, like all tests in this project, found in 'test-functions.cpp'.

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 $\log_2$.

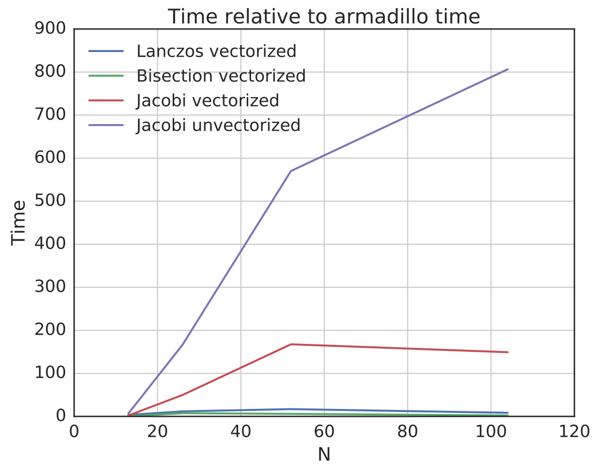

Figure 1: Algorithm times divided by Armadillo time. *Vectorization reduces times drastically, and the Sturm-Bisection and the Lanczos method is almost as fast as Armadillo.*

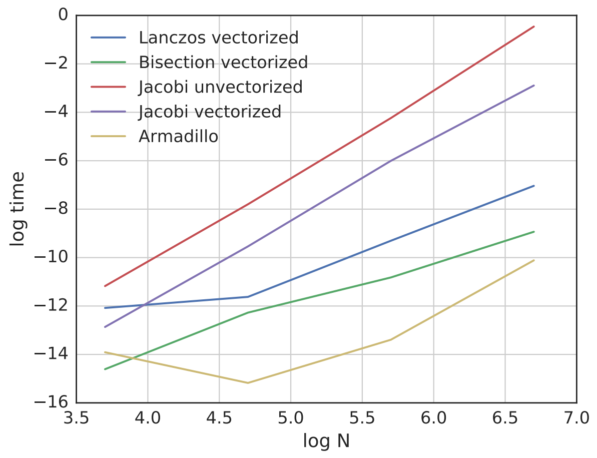

Figure 2: log2 Algorithm times. *Jacobi is third order, while Sturm-Bisection is seccond order.*

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.

For Lanczos, it is only to two highest N's that should be analyzed. There was no convergence for the other N's.

From the figure to the right we see that the order for Jacobi when it comes to time, looks like $\mathcal{O}(N^3)$. This was expected. Assuming the running time is proportional to the FLOPS, based on Hjorth-Jensen [2] 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.

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

We note that it is only the largest N's that is valid in also the right plot for Lanczos. The figure suggests that Lanczos is of almost the same order as Jacobi when it comes to FLOPS, but this is misleading. In principle, the Lanczos algorithm could stop at approximately the same number of iterations when N is doubled from a value where the eigenvalues have converged. This can be seen in the cases where ω_r and ρ_{max} are large.

3.3 Sturm bisection

The Sturm-Bisection implementation is found 'eigenvalueBisection.cpp'.

All part of the Sturm-Bisection family of functions are tested in the file 'test-functions.cpp'. All tests pass.

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

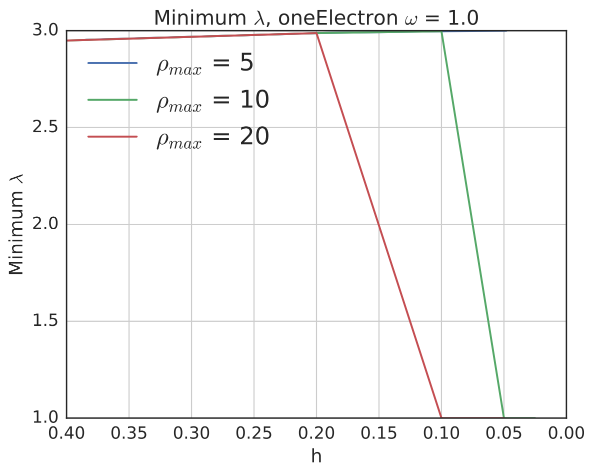

Figure 3: Minimum eigenvalue. One electron Sturm Bisection.
We get overflow for $N = 200$, which happens for the largest ρ_{max}'s

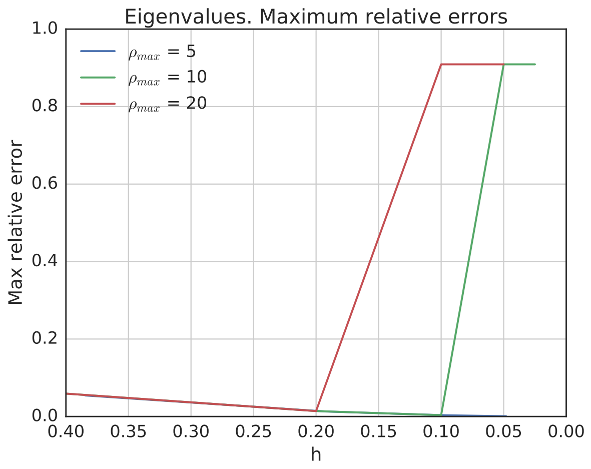

Figure 4: Maximum relative error of three smallest eigenvalues. One electron. Sturm Bisection.
Overflow for $N = 200$.

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. [1] 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.

3.4 Lanczos eigenvalues

The Lanczos algorithm is implemented in the program 'lanczos.cpp'. It is testet inside the program 'test-functions.cpp'. When applying Armadillo to compute the extremal eigenvalues for the tridiagonal matrix, it passes one of two tests. We are not sure why one test fails.

In addition, when running the tests,the prints show that the tridiagonal matrix produced by Lanczos' algorithm is in fact tridiagonal and that the matrix of Lanczos vectors is in fact orthogonal. .

In the theory section we mentioned that our case with a tridiagonal matrix, a part of the Lanczos algorithm can be calculated with considerble FLOP reduction. This method, toghether with the brute force method, is tested with success in 'testing-functions.cpp'. In our calculations we apply the alternative method successfully.

One important thing to note, is that we had to modify the algorithm slightly from e.g. Hjort-Jensen [2]. To get good results with the algorithm, we had to include a zero row in the Q-matrix. The result of this was that the first eigenvalue always is zero, while the proceeding eigenvalues equalled what should have been the correct eigenvalues starting from index 0. We have no idea why this change had to be made. Implementing the algotithm exactly as outlined in the books, did not give good results.

We have not reported the numbers on this, but nontheless it is important: Lanczos algorithm for the most part iterates to half of N before convergence is reached. Alse when ρ_{max} and ω_r is large, the algorithm starts to shine. In these cases the number of iterations needed for convergence is around a fourth of N.

Because of its speed, we use Lanczos' method for almost all plots in the following.

3.5 Comparison with theory

3.5.1 Jacobi

To see that our Jacobi-implementation is correct, the below figure shows the results for the one electron case when 'jacobi.cpp' is used.

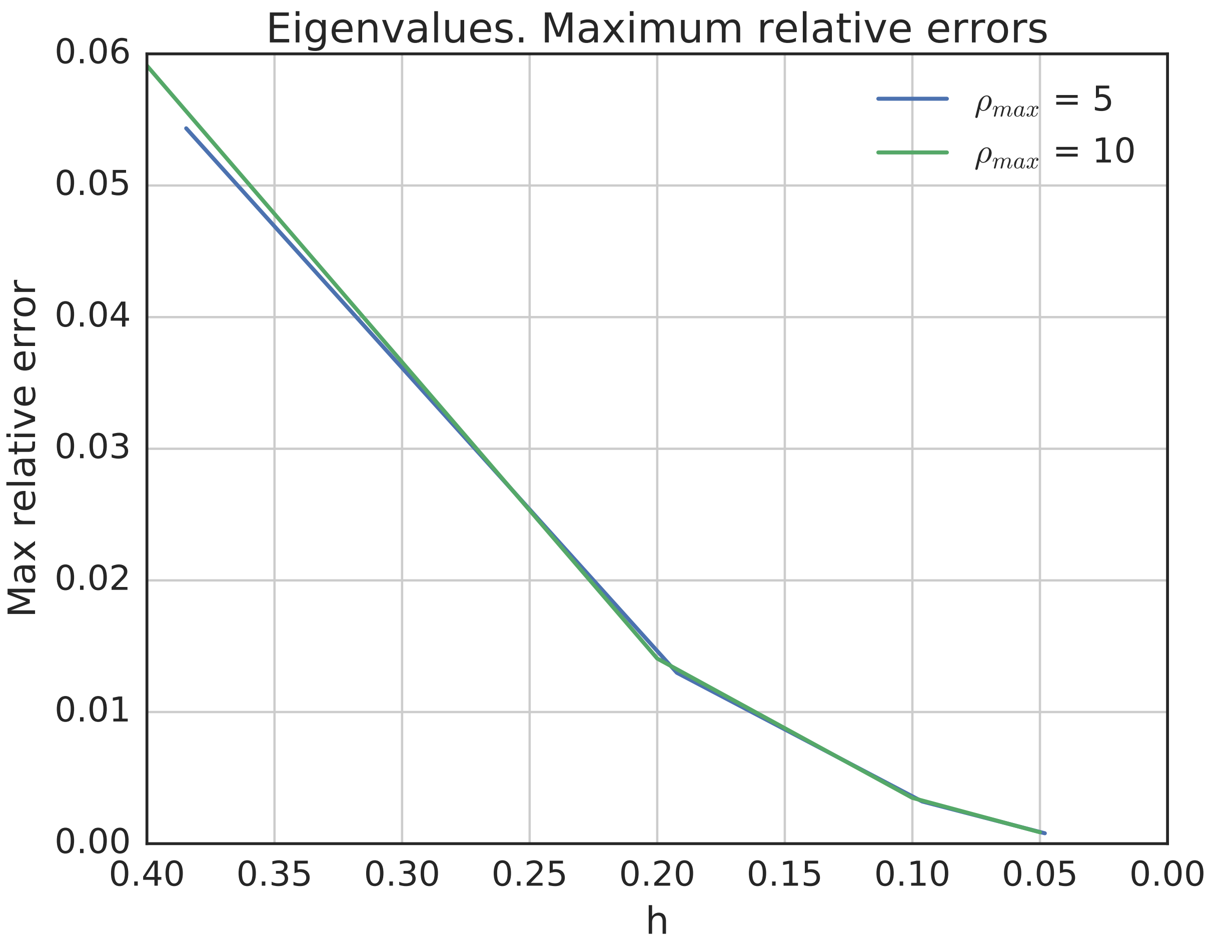

Figure 5: Maximum relative error of the three smallest eigenvalues, one electron. Jacobi.
Jacobi reproduces exact solution for the three lowest eigenvalues.

The above figure shows that our Jacobi implementation works, since the maximum relative error of the three smallest eigenvalues goes to zero. Also, the above figure suggests that Jacobi is not sensitive to overflow in the same way as the Sturm-Bisection algorithm. In the above figure $N = 200$ for the largest ρ_{max}, and it was for this N that the Sturm-Bisection method stopped working.

3.5.2 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 test for the convergence of the eigenvalues for different ρ_{max} in python (where we run everything from). This test 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 below four figures shows the results for four different ω's in the two electron repulsion case.

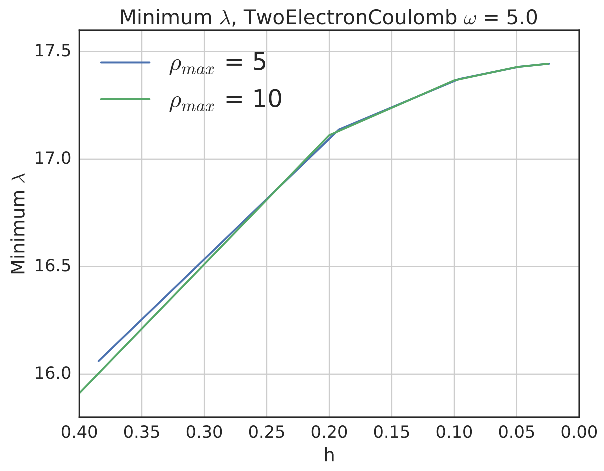

Figure 6: Comparison rhoMax. Minimum eigenvalue. Coulomb.
$\rho_{max} = 5$ *sufficient.*

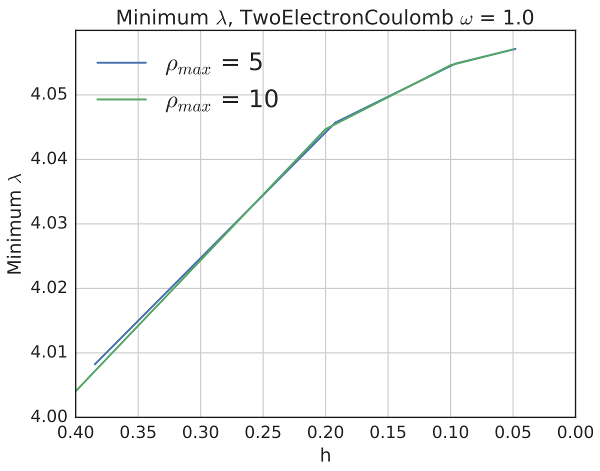

Figure 7: Comparison rhoMax. Minimum eigenvalue. Coulomb.
$\rho_{max} = 5$ *sufficient.*

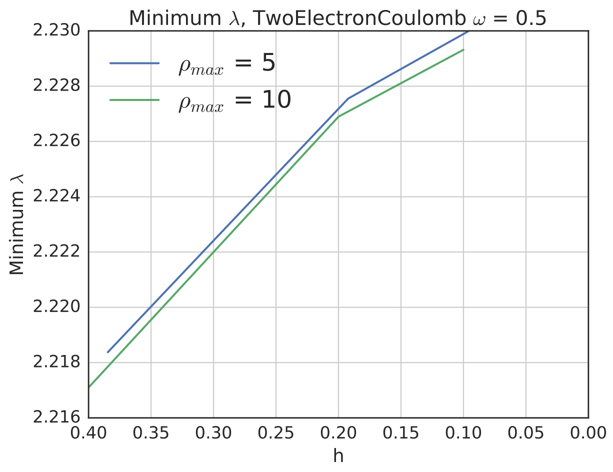

Figure 8: Comparison rhoMax. Minimum eigenvalue. Coulomb.
$\rho_{max} = 5$ *sufficient.*

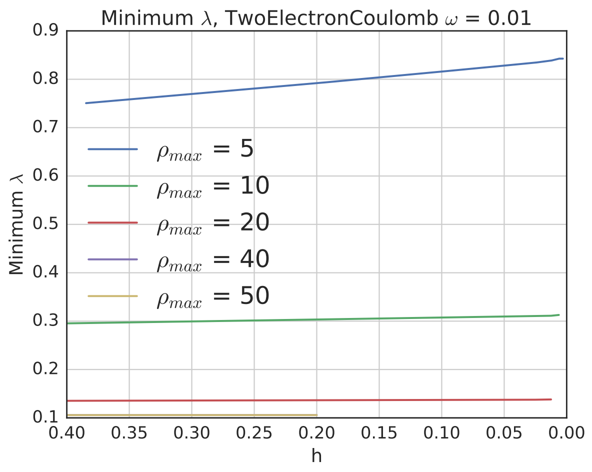

Figure 9: Comparison rhoMax. Minimum eigenvalue. Coulomb.
Much higher ρ_{max} *needed.*

The above figures shows that larger ρ_{max} is needed for convergence when ω gets small. Don't be fooled by figure 8, which by a quick look seems like showing that there is not convergence. When taking the scale into account, one can see that there is convergence. We typically require the minimum eigenvalue to change by less than 0.1 per cent. 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.

3.5.3 Single electron

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

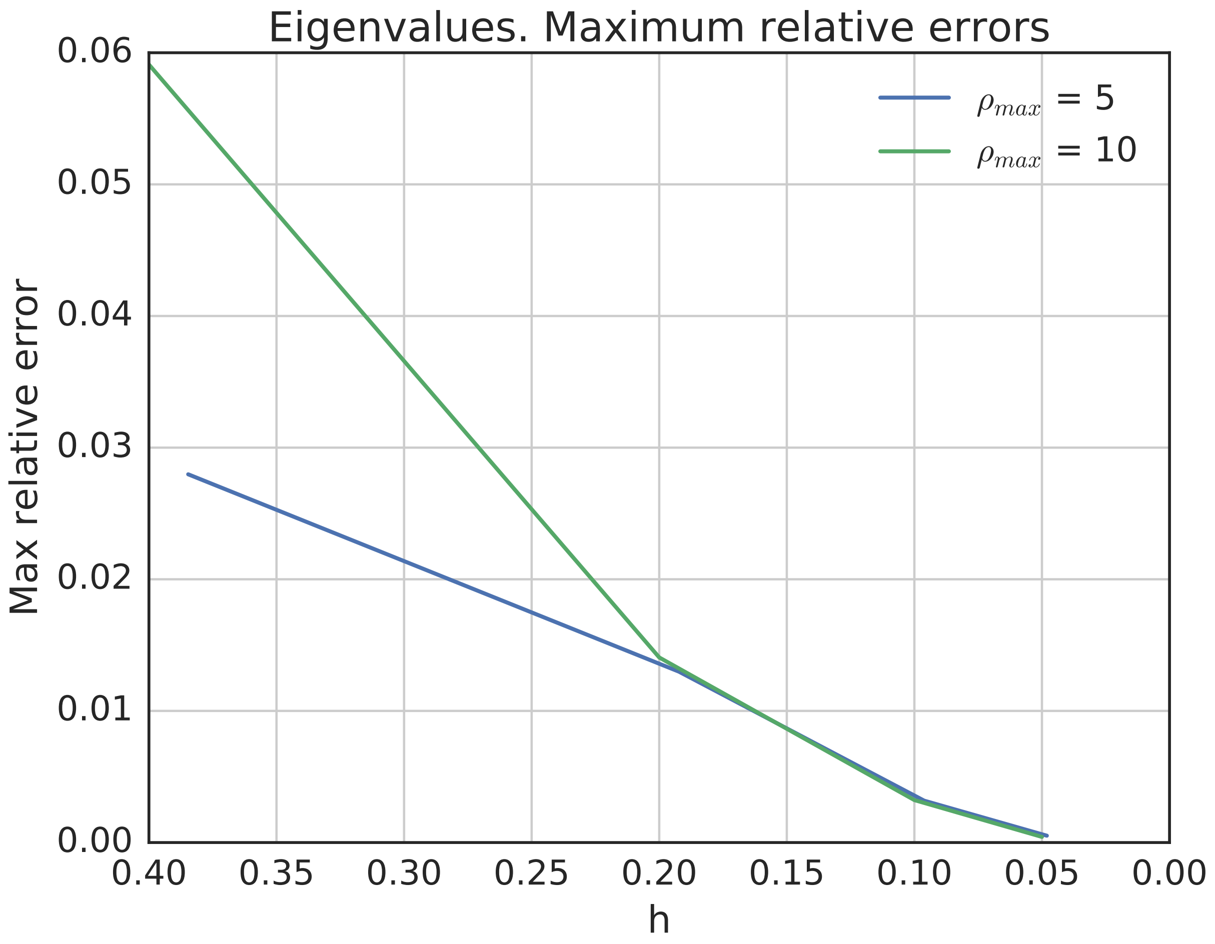

Figure 10: Maximum relative error of the three smallest eigenvalues, one electron.
Anaytical result reproduced for first ρ_{max}.

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

3.5.4 Two electrons and no repulsion

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

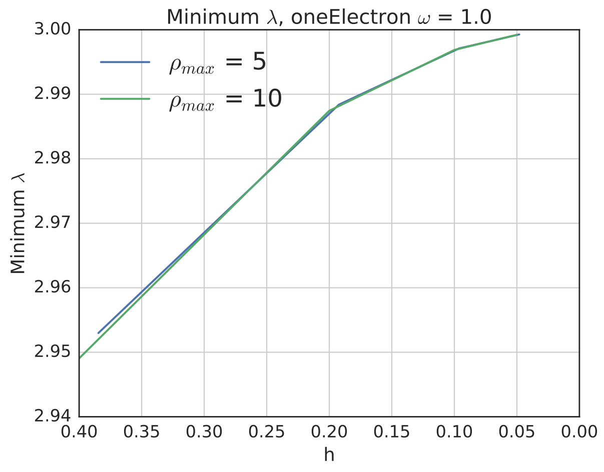

Figure 11: Minimum eigenvalue. 1 electron.

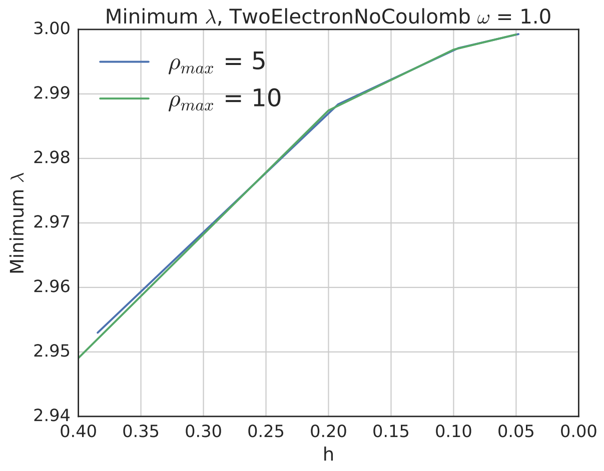

Figure 12: Minimum eigenvalue. 2 electrons no Coulomb.
The solution equals to single electron solution in the figure to the left.

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.

3.5.5 Two electron and repulsion

For the case with $\omega_r = 0.25$ we can compare our results with the analytical results from Taut (1993) [5]. 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.

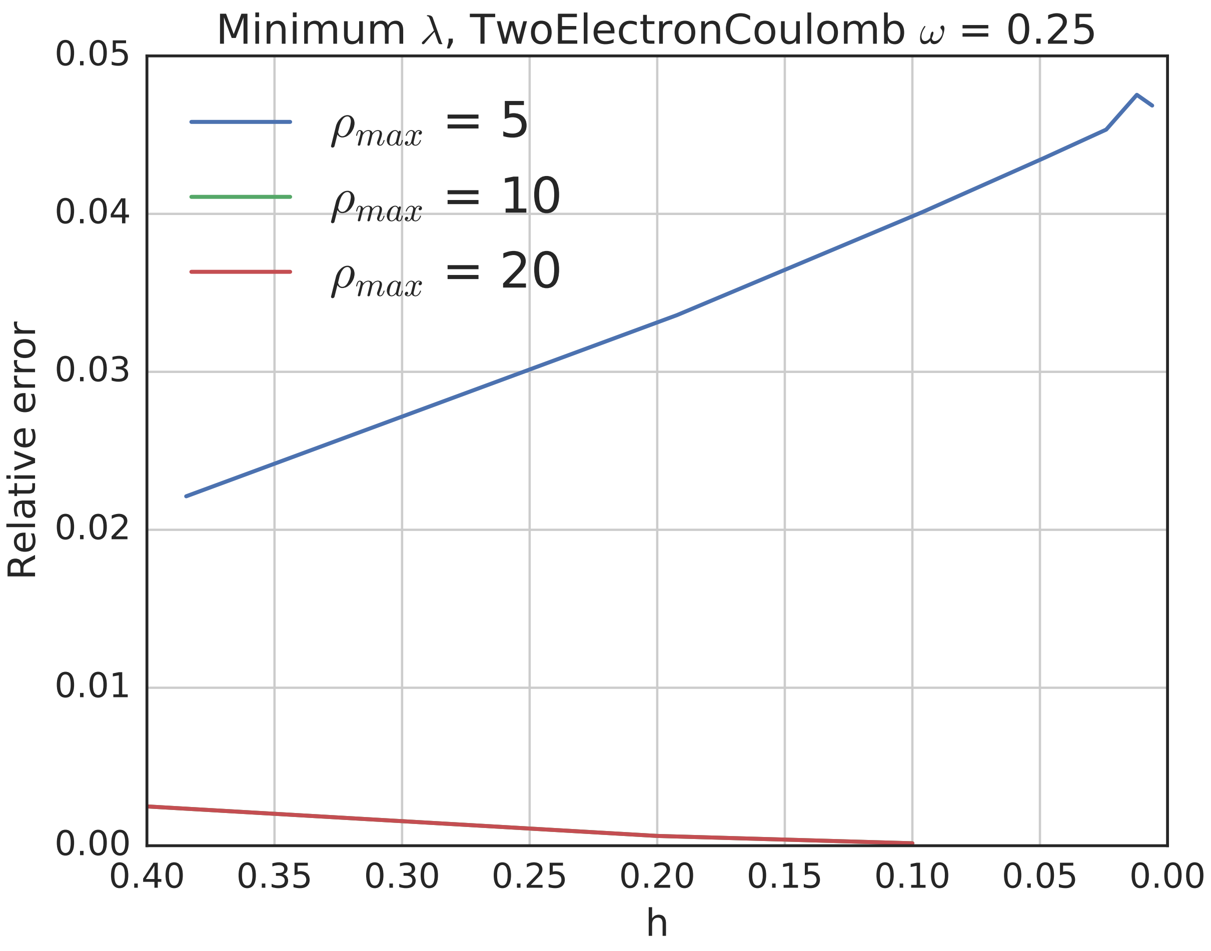

Figure 13: Relative error smallest eigenvalue.
Result from Taut (1993) [5] reproduced.

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.

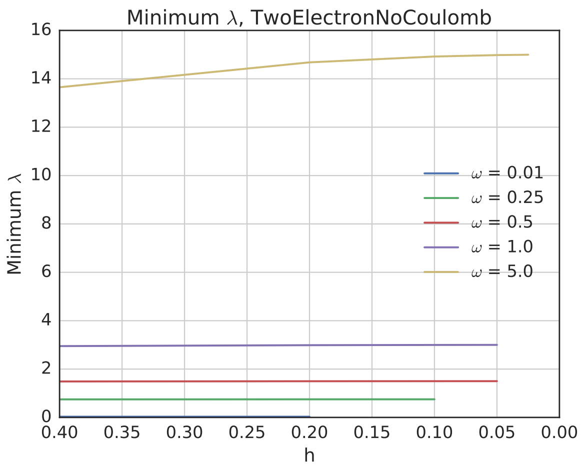

Figure 14: Minimum eigenvalue. 2 electron. No Coulomb.
The eigenvalues increases with ω_r.

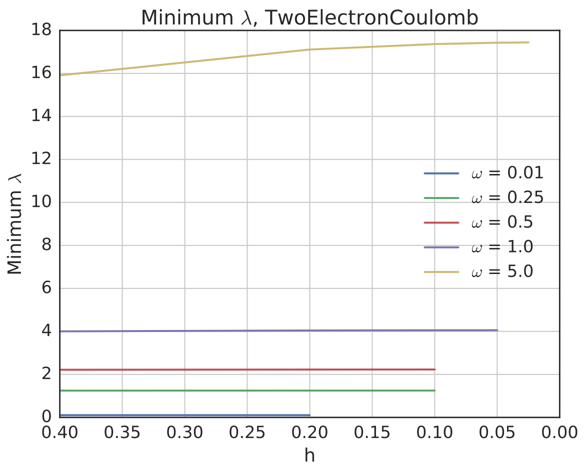

Figure 15: Minimum eigenvalue. 2 electrons Coulomb.
Adding Coulomb-force increases the eigenvalues. Compare with figure to the left.

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

4 Conclusions

In this project we have solved three different physical eigenvalue problems 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. Also it reduced the number of simulations required.

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

We applied three different algorithms for solving the linear system: the Jacobi method, the Sturm-Bisection method, and Lanczos' method. The last two methods were drastically faster than the Jacobi method, which was very slow. However, the Sturm-Bisection method was shown to be prone du overflow issues when the matrix got big. Lanczos algorithm did on the other hand, when implemented with a slight twist, prove to be both fast and accurate.

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.

Finally a few words about short-commings. It should be mentioned that there are versions of Jacobi's method for eigenvalues that are more efficient than the method applied in this report. 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.

In the Sturm-Bisection method we could have used the information about the tridiagonlity of the matrix more efficiently by calculating two arrays, one for the diagonal and one for the non-diagonal, instead of calculating the full matrix. Having done this, we could have sent in the two arrays instead of the full matrix to our Sturm-Bisection family of functions. This would be less memory demanding, since one would only have to store two N-dimensional arrays instead of an NxN matrix.

Also in the Lanczos method we could have used the fact the A was tridiagonal to an even greater extent than we did. E.g. the computation of the tridiagonal matrix, $Q^T AQ$ could have utlized the tridiagonal form in the same way we utilized the tridiagonal form of A when calculating α.

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 Piazza 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.
- We were not able to understand the revised Sturm-Bisection algorithm from Barth et al.'s [1] paper on the revised Sturm-Bisection.
- Apart from the small details above, we are very happy about this project. How would have thought linear algebra could be fun?!

6 Bibliography

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

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

[3] Hjorth-Jensen, M.(2017) Project 2, fys4150 2017. `https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Projects/2017/Project2/pdf/Project2.pdf`

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

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