Eigenvalue problems, from the equations of a buckling beam to Schroedinger's equation for two electrons in a three-dimensional harmonic oscillator well

Astrid Bragstad Gjelsvik, Erlend Andreas Longva and Karianne Strand*
Department of physics, University of Oslo, P.O. Box 1048 Blindern, N-0316 Oslo, Norway

ABSTRACT
In this project we have looked at numerical methods for solving different eigenvalue problems and how different physical problems can be formulated as eigenvalue problems, using scaling. We have solved a buckling beam problem, as well as Schroedinger's equation for one electron in a harmonic oscillator potential, using eigenvalue solvers. The methods we have used to solve for eigenvalues are Jacobi's method and the bisection method for tridiagonal matrices, which we have compared to the c++ library Armadillo's function `eig_sym`. We have found that the computation time for Jacobi's method differ significantly from the two latter ones for large matrices. For a 100×100 matrix, the computing time of Jacobi's method was found to be 0.455277 s, while it was found to be 0.006756 s for the bisection method and 0.005511 s for `eig_sym`.

I. INTRODUCTION

Many physical problems can, through scaling of equations, be interpreted as eigenvalue problems. As physicists, we are therefore highly interested in developing accurate and computationally economical eigenvalue solvers. However, there are many different ways to solve eigenvalue problems numerically. Here, we evaluate a few of them.

In this project we have embarked on two physical problems. These are a buckling beam problem, and a Schroedinger equation for one electron in a harmonic oscillator potential. These problems are particularly relevant for this project as they have analytical eigenvalues we can compare against our computed ones. We solve these problems by scaling them and setting them as eigenvalue problem for tridiagonal matrices. We apply different eigenvalue solvers, and compare the different computation times. The solvers we apply are one based Jacobi's method, and one based on the bisection method. We also compare these to the c++ library armadillo's function `eig_sym`.

In the following we will first go through the methods we used in our project. We will explain the theory behind Jacobi's method and the bisection method, and how we implement these. We will also explain how we scale the buckling beam equation and the Schroedinger equation, and how we implemented them. We will then present the results of our project. Here we will present the computation time for the different methods, as well as the eigenpairs computed compared with the analytical ones. Finally, we will discuss the advantages and disadvantages of the different eigenvalue solvers.

* astridbg@student.matnat.uio.no
erlenalo@student.ifikk.uio.no
kastr@student.matnat.uio.no

II. METHOD

In this project our goal is to explore methods for solving eigenvalue problems, as well as formulating physical problems so that they can be represented as eigenvalue problems. We write the physical problems on the form of tridiagonal matrices. We use that fact that our problems have known analytical eigenvalues to check if the diagonalization function `eig_sym` of the c++ library armadillo returns the same values. In addition, we implement the Jacobi rotation algorithm to find eigenvalues, and compare these to the analytical ones. We also implement the bisection method to find the eigenvalues. We are then able to compare the computation time of Jacobi's method to the armadillo function `eig_sym` and the bisection method.

In the following we will first describe how Jacobi's method can be used to find the eigenvalues of a symmetric matrix **A**. We will then see how our our buckling beam problem can be scaled and put on the form of an eigenvalue problem, and how we can implement Jacobi's method to solve it. We will then show how we can do the same for Schroedinger's equation for electrons in a three-dimensional harmonic oscillator potential. Finally, we will show how the bisection method can be implemented. For all program code, tests, output files and plots obtained see our GitHub-repository.

A. Jacobi's method

In Jacobi's method we perform similarity transformations on a real and symmetric matrix **A** until it is practically diagonal. This means that we continue to perform similarity transformations until all the non-diagonal elements have been transformed to values less than some small value ϵ. We use $\epsilon = 10^{-8}$.

A similarity transfrom **B** of a symmetric matrix **A** can be described by

$$\mathbf{B} = \mathbf{S}\mathbf{A}\mathbf{S}^T \tag{1}$$

where $\mathbf{S}$ is some transformation matrix. This matrix is orthogonal and unitary so that $\mathbf{S}^T\mathbf{S} = \mathbf{S}^{-1}\mathbf{S} = \mathbf{I}$. In an eigenvalue problem $\mathbf{A}\mathbf{u} = \lambda\mathbf{u}$, we can acquire this similarity transformation by multiplying with $\mathbf{S}$ on both sides and inserting $\mathbf{S}^T\mathbf{S} = \mathbf{I}$ between $\mathbf{A}$ and $\mathbf{u}$. We will then get

$$\mathbf{S}\mathbf{A}\mathbf{S}^T\mathbf{S}\mathbf{u} = \mathbf{S}\lambda\mathbf{u} \tag{2}$$
$$\mathbf{B}\mathbf{S}\mathbf{u} = \lambda\mathbf{S}\mathbf{u} \tag{3}$$

where $\mathbf{S}\mathbf{u}$ is the transformed eigenvector. Under such a transformation, we can show that the orthogonality and norm of the eigenvectors of $\mathbf{A}$ will be preserved. If we assume an orthogonal set of eigenvectors $\mathbf{u}_i$ such that

$$\mathbf{u}_i^T\mathbf{u}_j = \delta_{ij} \tag{4}$$

and perform a transformation to a set of vectors $\mathbf{w}_i = \mathbf{S}\mathbf{u}_i$ where $\mathbf{S}$ is an orthogonal, unitary matrix, we will have

$$\mathbf{w}_i^T\mathbf{w}_j = (\mathbf{S}\mathbf{u}_i)^T\mathbf{S}\mathbf{u}_j = \mathbf{u}_i^T\mathbf{S}^T\mathbf{S}\mathbf{u}_j = \mathbf{u}_i^T\mathbf{u}_j = \delta_{ij} \tag{5}$$

using that $\mathbf{S}^T\mathbf{S} = \mathbf{I}$.

In Jacobi's method we determine the transformation matrix $\mathbf{S}$ is such a way that it rotates the matrix with an arbitrary angle θ in the n-dimensional space. Our $\mathbf{S}$ will in other words look something like this[1]:

$$\mathbf{S} = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & \cos\theta & 0 & \dots & 0 & \sin\theta \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & -\sin\theta & \dots & \dots & 0 & \cos\theta \end{bmatrix} \tag{6}$$

We choose θ so that for each rotation we set the largest non-diagonal element to zero [1]. We choose the largest element in order to reduce the number of iterations needed, and diagonalize the matrix faster. We find the largest off-diagonal matrix element a_{kl} of $\mathbf{A}$ using the following algorithm:

```
double max;
int k,l;
max = 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));
      k = i; l = j;}}}
```

Since the matrix is symmetric, we only have to find the maximum off-diagonal element on one side of the diagonal. Here we do it on the upper side.

We determine the angle θ by requiring that we have

$$b_{kl} = a_{kl}(\cos^2\theta - \sin^2\theta) + (a_{kk} - a_{ll})\cos\theta\sin\theta = 0$$
$$1 - \tan^2\theta + \frac{a_{kk} - a_{ll}}{a_{kl}}\tan\theta = 0$$
$$\tan^2\theta - 2\tau\tan\theta - 1 = 0$$

where $\tau = (a_{kk} - a_{ll})/2a_{kl}$. We are then able to determine $\tan\theta$ by

$$\tan\theta = -\tau \pm \sqrt{1+\tau^2} \tag{7}$$

which allows us to determine $\cos\theta = 1/\sqrt{1+\tan^2\theta}$ and $\sin\theta = \cos\theta\tan\theta$. When determining θ we generally want to choose the smallest θ for the rotation, in order to change already small matrix elements as little as possible [1]. However, when computing equation 7 we risk getting numbers very close to zero, and thereby risk losing numerical precision [1]. We drive a_{kl} to zero, which makes τ large, and so $\sqrt{1+\tau^2} \approx \tau$. To avoid this, we multiply and divide with $\tau \mp \sqrt{1+\tau^2}$, giving us:

$$\tan\theta = \frac{(-\tau \pm \sqrt{1+\tau^2})(-\tau \mp \sqrt{1+\tau^2})}{-\tau \mp \sqrt{1+\tau^2}} \tag{8}$$
$$= \frac{\tau^2 - (1+\tau^2)}{-\tau \mp \sqrt{1+\tau^2}} = \frac{-1}{-\tau \mp \sqrt{1+\tau^2}} \tag{9}$$
$$= \frac{1}{\tau \pm \sqrt{1+\tau^2}} \tag{10}$$

To get the smallest angle θ, without losing numerical precision, we therefore write:

```
if( tau > 0 ){
  t = 1./(tau + sqrt(1 + tau*tau));
}
else{
  t = 1./(tau - sqrt(1 + tau*tau));
}
```

where t $= \tan\theta$ and tau $= \tau$.

The implementation of the rotation of the matrix then proceeds as follows:

```
double a_ik, a_il, a_kk, a_ll, a_kl;
a_kk = A(k,k); a_ll = A(l,l); a_kl = A(k,l);
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(i,l) = a_il*c + a_ik*s;
    A(k,i) = A(i,k);
    A(l,i) = A(i,l);
  }
}
A(k,k) = a_kk*cc - 2*a_kl*cs + a_ll*ss;
A(l,l) = a_ll*cc + 2*a_kl*cs + a_kk*ss;
A(k,l) = 0.0;
A(l,k) = 0.0;
```

Here `c` and `s` denotes $\cos\theta$ and $\sin\theta$. These rotations are repeated until the maximum off-diagonal element a_{kl} is less than ϵ. We perform the diagonalization for different matrix dimensionalities and count the number of rotations, or similarity transformations, needed for each matrix. This relationship is shown in the result section.

Given an eigenvector matrix $\mathbf{R}$, we can find out how the eigenvectors change under the similarity transformations at the same time as we find the transformed matrix elements of $\mathbf{A}$. The transformed eigenvectors are, as we saw earlier, given by $\mathbf{Su}$. For each transformation we will therefore have [1]:

```
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;
```

where `R` will be tranformed to a new matrix of eigenvectors.

B. The buckling beam problem

We now want to solve the two-point boundary value problem of a buckling beam, using Jacobi's method. To do so we will need to express the buckling beam problem in terms of an eigenvalue problem. Starting off with the one dimensional classical wave equation on differential form, we have that

$$\gamma\frac{d^2u(x)}{dx^2} = -Fu(x), \tag{11}$$

for a beam of length L placed along the x-axis. Here $u(x)$ is the displacement of the beam in the y-direction, γ is the spring constant, and F is the force applied to the end of the beam at $(L, 0)$. The Dirichlet boundary conditions are then $u(0) = u(L) = 0$. Furthermore we can scale equation 11 using the dimensionless variable

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

where $\rho \in [0, 1]$. Thus, when scaling equation 11 we get

$$\frac{d^2u(\rho)}{dx^2} = -\lambda u(\rho), \tag{12}$$

where $\lambda = FL^2/\gamma$. Next we define the step length h such that

$$h = \frac{\rho_N - \rho_0}{N}, \tag{13}$$

where N is the number of steps, and ρ_N and ρ_0 are the maximum and minimum values of ρ respectively. With that step length we can find ρ at each step i, such that

$$\rho_i = \rho_0 + ih \quad \text{where} \quad i = 1,\ 2,\ ...,\ N. \tag{14}$$

If we now discretize equation 12 using the step length from equation 13 and ρ_i in equation 14 we get that

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

which further can be written as

$$-\frac{u_{i+1} - 2u_i + u_{i-h}}{h^2} = \lambda u_i. \tag{15}$$

Equation 15 can then be written as an eigenvalue problem, such that

$$\begin{bmatrix} d & a & 0 & 0 & \dots & 0 & 0 \\ a & d & a & 0 & \dots & 0 & 0 \\ 0 & a & d & a & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & a & d & a \\ 0 & \dots & \dots & \dots & \dots & a & d \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \dots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = \lambda \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \dots \\ u_{N-2} \\ u_{N-1} \end{bmatrix}.$$

This eigenvalue problem has analytical eigenpairs, with eigenvalues given as

$$\lambda_i = d + 2a\cos\left(\frac{j\pi}{N+1}\right) \tag{16}$$

and eigenvectors

$$\mathbf{u}_j = [\sin\left(\frac{j\pi}{N}\right), \sin\left(\frac{2j\pi}{N}\right), \dots, \sin\left(\frac{(N-1)j\pi}{N}\right)] \tag{17}$$

for $j = 1, 2, \dots, N$ [2].

Our goal for this part of the project is to put our implementation of the Jacobi method to the test. Therefore our first task is to check that the matrix we derived for the buckling beam problem is correct. To do this we compute the eigenvalues using the armadillo function eigsym, which gives us both eigenvalues and eigenvectors for the matrix, and compare these to the analytical ones. A plot comparing these is presented in the results section.

Now that we can be certain the matrix is set up correctly we test our Jacobi method against the armadillo eigsym function. First we made a script that checks the number of similarity transformations needed in the Jacobi method to obtain precise eigenvalues. To check the precision we compare the eigenvalues with those obtained with armadillo. We also compare the eigenvector for the lowest eigenvalue and compare this with the one obtained with armadillo. We could also have compared this directly with the analytical one, but since armadillo has already been compared with the analytical values we do not have to do this. The resulting plots are presented in the results section.

C. Schroedinger's equation for one electron

scripts/ In this part of the project we model the radial part of Schroedinger's equation for one electron. The goal is to set it up as an eigenvalue problem by rescaling as we did for the buckling beam. We start from the radial Schroedinger's equation for one electron given on the form:

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

In this problem we will use the harmonic oscillator potential $V(r)=(1/2)m\omega^2r^2$, and thus the energies or eigenvalues (unscaled) for our problem are:

$$E_{nl}=\hbar\omega\left(2n+l+\frac{3}{2}\right) \tag{18}$$

For the sake of simplicity we restrict ourselves to the case of $l=0$, recalling that l is the orbital momentum quantum number of the electron. Making the substitution $R(r)=(1/r)u(r)$ we now have:

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

We then rescale the problem by introducing the dimensionless variable $\rho=(1/\alpha)r$ analogous to our method for the buckling beam problem. Inserting the explicit potential $V(\rho)=(1/2)m\omega^2\alpha^2\rho^2$ we obtain the following expression.

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

Now multiplying everything with $2m\alpha^2/\hbar^2$ we get:

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

With $m\omega^2=k$. We then fix α so that $(mk/\hbar^2)\alpha^4=1$. Thus we define:

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

Finally we get the Schroedinger equation on the form:

$$-\frac{\mathrm{d}^2}{\mathrm{d}\rho^2}u(\rho)+\rho^2u(\rho)=\lambda u(\rho)\ . \tag{19}$$

Now we discretize the solution. The steplength h is found as before $h=(\rho_{max}-\rho_0)/N$ where N is the number of steps or dimensionality of our matrix. Since ρ is the radial coordinate $\rho_{max}\to\infty$ so we must experiment with values of ρ_{max} that gives us a good approximation.

We can then rewrite the Schroedinger equation totally analogous to the buckling beam, but with the added Coulomb potential along the diagonal. With d_i being on the diagonal, and e_i being on the two off diagonals we have:

$$d_i=\frac{2}{h^2}+V_i \quad \text{, where} \qquad V_i=\rho_i^2=(\rho_0+ih)^2$$
$$e_i=-\frac{1}{h^2}.$$

Our goal is then to find a good approximation for the eigenvalues. The analytical eigenvalues are $\lambda_i=3,7,11,15....$ We implement an algorithm that solves the matrix we have derived using the jacobi method we have developed earlier in the project. This is not optimal since we would like to increase N and ρ_{max} substatially and the jacobi method is quite slow. We set the algorithm up to solve for increasing values of N with a static value of ρ_{max}. For each N we compare the obtained eigenvalues with the analytical eigenvalues and compute an array of the relative error between each eigenvalue. We then find the mean deviation in the array of relative error, as well as the standard deviation. This is then saved in a file which is sent to a python program that plots this. The N values are increased logarithmic.

We also find and plot the eigenvalues for the problem using the jacobi method and armadillos eigsym function and compare these. Additionally the eigenvector corresponding to the smalles eigenvalue is plotted. As always, all plots are presented in the results section.

D. Bisection method

Another strategy we can apply in order to solve eigenvalue problems is the bisection] method. In this strategy, unlike in Jacobi's method, we take advantage of the fact that our matrix is tridiagonal, something which makes the algorithm more efficient.

The method is based on finding the roots of the characteristic polynomial of our matrix $\mathbf{T}$, which equal the eigenvalues of the matrix, using bisection. The problem we want to solve is in other words essensially this:

$$P_N=\det(\mathbf{T}-\lambda\mathbf{I})=0 \tag{20}$$

where

$$P_N=\begin{vmatrix} d_1-\lambda & a_2 & 0 & \dots & 0 & 0\\ a_2 & d_2-\lambda & a_3 & \dots & 0 & 0\\ \dots & \dots & \dots & \dots & \dots & \dots\\ 0 & \dots & \dots & a_{N-2} & d_{N-2}-\lambda & a_{N-1}\\ 0 & \dots & \dots & \dots & a_{N-1} & d_{N-1}-\lambda \end{vmatrix} \tag{21}$$

The bisection method of finding roots of a function is generally applied by determining an interval where one knows that a root is located, and evaluating the function in the middle of that interval. If the sign of the function changes between the start of the interval and the middle, there is a root located in the first half of the interval (and similarily in the second half if the sign changes between the middle and the end). One can then change the interval of interest to the first half, and evaluate the function again in the middle of this interval. This procedure can be repeated until the interval is small enough around the root so that we know the root with a sufficiently small uncertainty.

The interval where the roots of our characteristic polynomial is located is given by Gershgorin's theorem. All the eigenvalues are contained in the union of the N intervals $d_i \pm (|a_i| + |a_{i+1})$, with $a_0 = a_N = 0$ [3]. This can give us an interval (x_{min}, x_{max}) where we know that all our eigenvalues are contained.

$$x_{min} = \min\{d_i - (|a_i| + |a_{i+1}|)\} \tag{22}$$

$$x_{max} = \max\{d_i + (|a_i| + |a_{i+1}|)\} \tag{23}$$

The problem we face with our $\mathbf{T} \in \mathbb{R}^{N\times N}$ is that it is not straightforward to evaluate P_N. However, we can evaluate it via a sequence of polynomials with increasing degree. This is what we call a Sturm sequence. Consider the upper left element of $\mathbf{T}$. This gives us a first-order polynomial

$$P_1(\lambda) = d_1 - \lambda \tag{24}$$

which we can solve for $P_1 = 0$ to get the root of this first-degree polynomial $\lambda_1^{(1)} = d_1$. If we then consider the determinant of the 2×2 matrix in the upper left corner of $\mathbf{T}$, we get a second-order polynomial

$$P_2(\lambda) = (d_1 - \lambda)(d_2 - \lambda) - a_2^2 \tag{25}$$

which we can express as

$$P_2(\lambda) = (d_2 - \lambda)P_1 - a_2^2 P_0 \tag{26}$$

using the trivial $P_0 = 1$, and solve for $P_2 = 0$ to get the second-degree roots $\lambda_1^{(2)}$ and $\lambda_2^{(2)}$. In fact, we can express the characteristic polynomial of each sub-matrix of $\mathbf{T}$ in terms of the two previous ones in the following way:

$$P_i = (d_i - \lambda)P_{i-1} - a_i^2 P_{i-2}, \quad i = 2, \ldots, N \tag{27}$$

The roots of the polynomials P_i will be interlacing in such a way that

$$\lambda_k^{(i+1)} < \lambda_k^{(i)} < \lambda_{k+1}^{(i+1)} \tag{28}$$

where i is the polynomial degree and k is the eigenvalue rank. We are here assuming that our eigenvalues are sorted so that

$$\lambda_1^{(i)} < \lambda_2^{(i)} < \cdots < \lambda_N^{(i)} \tag{29}$$

If the sign of P_i is different from P_{i-1} for some value x_0, we know that there must be one more root for P_i than for P_{i-1} for $\lambda < x_0$. If we perform this evaluation for P_i, $i = 2, \ldots, N$, we end up with the number of roots $\lambda_k < \lambda_0$ of P_N. This algorithm is implemented as follows:

```
c = 0; r = 1;
for (int i = 0; i < N; i++){
  if (r != 0){
    r = d(i) - xm - aa(i)/r;
  }
  else{
    r = d(i) - xm - aa(i)/nonzero;
  }
  if (r < 0){
    c += 1;
  }
}
```

Here `c` is counter of how many roots P_N has for $\lambda <$ `xm`, `d` is a vector of the diagonal elements, `aa` is vector of the squared off-diagonal elements. `r` is the ratio P_i/P_{i-1}, and is updated by using the previous `r` $= P_{i-1}/P_{i-2}$. If `r`< 0, we count this is as one root.

For each eigenvalue λ_k we want to find, we can then use bisection to find out which side of our middle point we have `c` $< k$, and continue to make the interval smaller around λ_k until the interval is smaller than a certain small value ϵ. We use $\epsilon = 10^{-8}$. We start by finding the largest eigenvalue, using the full interval (x_{min}, x_{max}), and as we move on to finding smaller eigenvalues λ_{k-1} we use the eigenvalues found in the previous round as an upper bound. If we through the process find that there are values under which there are no roots, or not as many roots as the number of roots we want to find, we can make this a lower bound for the next eigenvalues we want to find. If we find a value under which there are some roots, but not all we want to find, we can make this value an upper bound for the eigenvalues which we will find under it. This shortens the computing time of the bisection method, something which was found by Barth et. al. [3]. The implementation of the code can be seen below.

```
if (c < k + 1){
  if (c < m1 + 1){
    xl = lb(m1) = xm;
  }
  else{
    xl = lb(c) = xm;
    if (x(c-1) > xm){
      x(c-1) = xm;
    }
  }
}
else{
  xu = xm;
}
```

Here `xl` and `xu` are the lower bounds and upper bounds respectively for our eigenvalue. `k` is the index of the eigenvalue we are trying to find, while `m1` is the index of the lowest eigenvalue we want to find. Here we find all the eigenvalues. `lb` is a vector of lower bound for each eigenvector we want to find, while `x` is vector where we store eigenvalues, which also function as upper bounds for the lower eigenvalues.

E. Unit testing

Testing the programs is an integral part of developing code when doing computational science. In this project we have implemented some minor unit tests which checks that some parts of our code works. Since we have gone for the approach of setting up a library of functions which are then run through scripts it makes sense to set up a script going through our functions and testing each one independently. Thus we test each "unit" of our program. Since testing everything would be a bit excessive for this project we have limited our scope to testing the function responsible for finding the maximum value element of a symmetric matrix used in the jacobi method. We also test the jacobi method itself.

To test the maximum element function we set up two simple 5 times 5 matrices. The first has a positive maximum value in the upper right corner, the second has a negative value. This ensures that our program always picks the biggest absolute value of each element. We also set up the diagonal to contain values larger than the maximum off-diagonal element, this way we ensure that the function does not pick out diagonal elements. Our function only goes through the upper triangular matrix since we are only concerned with symmetric matrices in this project.

The second test is very similiar to what we are testing through out this project. It takes a static symmetric matrix and finds the eigenvalues using armadillo's eigsym function and then compares this with the jacobi method eigenvalues. If the difference is less than the threshold of the jacobi method used (we default to $\epsilon = 10^{-8}$) the test is passed. Both tests uses the standard "cassert" assert function to fail or pass the tests. The tests can be initialized through the main program found on github, how to run this and the other programs are explained in the readme file.

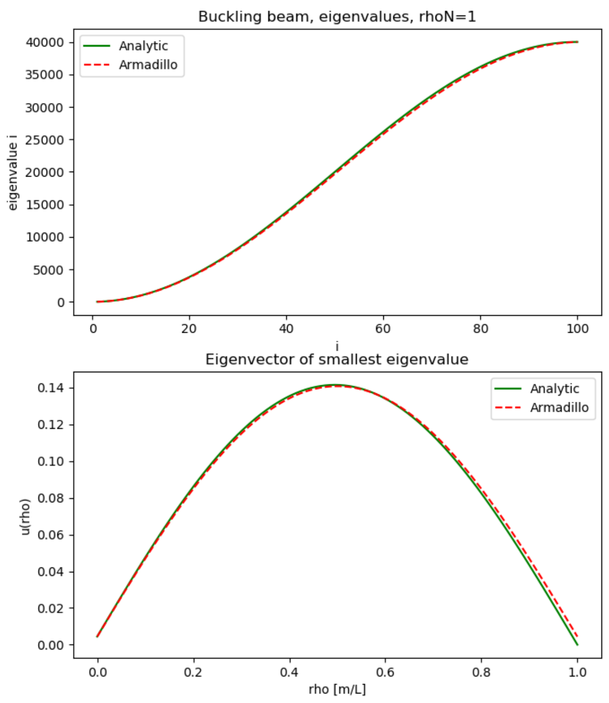

Figure 1. Comparison of eigenvalues and eigenvector for the smallest eigenvalue. Analytic results are plotted in green, while the values obtained from armadillo's eigsym are plotted in red.

III. RESULTS

A. The buckling beam problem

We first compared the analytically obtained eigenvalues with the eigenvalues obtained with armadillos eigsym function. We also compared the eigenvector for the lowest eigenvalue. The results are presented in figure 1. The results are quite good, but we can see some slight deviations for the eigenvalues in the middle of the plot. The eigenvector only deviates slightly when $\rho \to 1$.

Next we found the iterations or rather the similarity transformations needed to compute the eigenvalues using the jacobi method. The program looped over different dimensionality N and found the number of transformations needed. The results are presented on a logarithmic plot to show case how the number of transformations goes for N. A line fitted to N^2 is plotted alongside the results with a dashed red line. We also computed the CPU time for each value of N shown underneath with a blue line. The results are presented in figure 2.

The results from comparing the jacobi method eigenvalues and eigenvector for the lowest eigenvalue are presented in figure 3. Computing this we used a threshold $\epsilon = 10^{-8}$. The results show that the jacobi method provides very accurate results, but recalling the deviations between the analytical and armadillo's eigsym we observe that these also will apply for the results obtained with the jacobi method.

The last thing we do for the buckling beam problem is to compare the time usage of the different methods. In figure 4 we provide a comparison between the jacobi method and armadillo's eigsym function in obtaining the eigenvalues and vectors. It shows that the jacobi method is the most time ineffective of the two methods by a large margin. In table 1 CPU times for $N = 10, 100$ is shown for the jacobi method, eigsym and bisection is presented. The unstable nature of the values is likely due to CPU usage from other applications while running the simulation.

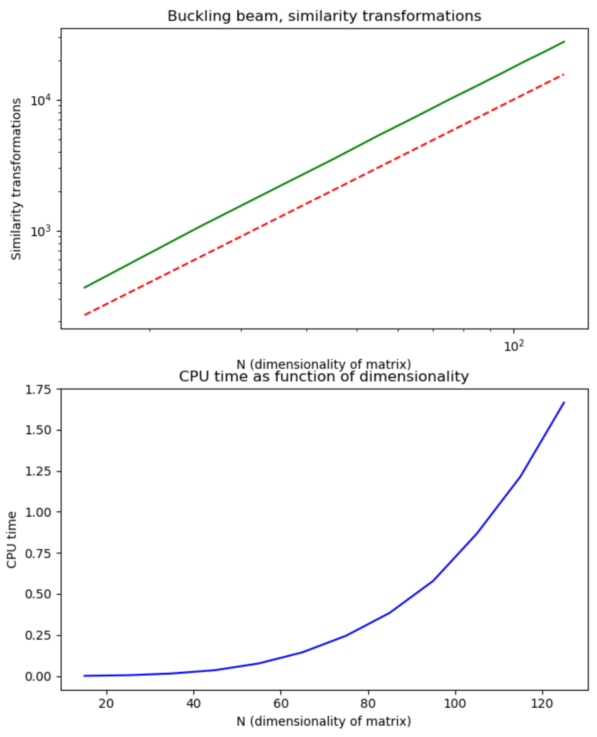

Figure 2. Logarithmic plot of similarity transformations for increasing dimensionality N (green), a line going as N^2 is included along side to show the similarity (dashed red line). Lower plot shows CPU time per value of N (blue).

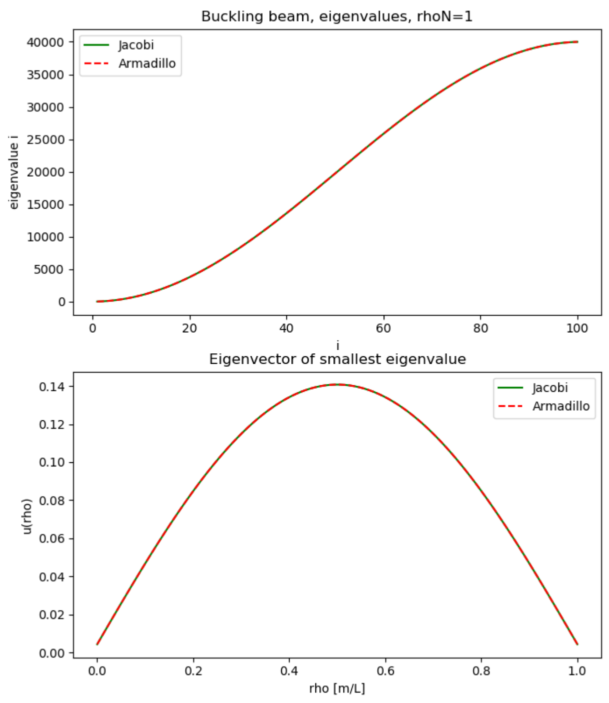

Figure 3. Comparison of eigenvalues and eigenvector for the smallest eigenvalue. The results from the jacobi method are plotted in green, while the values obtained from armadillo's eigsym are plotted in dashed red.

Table I. Computation Time [s] for Eigenvalue Solvers

N	Jacobi	`eig_sym`	Bisection
10	0.000153	0.000260	0.000240
100	0.455277	0.005511	0.006756

In table I we see the computation time in seconds for the different eigenvalue solvers we use in this project, applied for the buckling beam problem. The solvers are Jacobi's method, the armadillo function `eig_sym` and the bisection method.

We see that the methods have around the same computation time for a small matrix with dimension $N = 10$. In fact, here the Jacobi method is the fastest. However, the computation time for a larger matrix with dimension $N = 100$ is suddenly much larger for Jacobi's method than for the two other methods. For $N = 100$, we see that `eig_sym` function is the fastest, but the bisection is of the same order.

B. Schroedinger's equation for one election

The first result is plots of the eigenvalues and the eigenvector of the lowest eigenvalue obtained with the jacobi method and armadillo's eigsym function. We can see that the eigenvalues obtained using both methods are similar as expected, although the eigenvector from the eigsym function is negative. The results are shown in figure 5. For these plots the value of ρ_{max} was set to 40.

The plot showing the mean deviation in the relative error between numerical and analytical eigenvalues for the quantum dots problem is presented in figure 6. We observe that the mean deviation decreases when N increases but only up to the second last point. The bars on the plot represents the standard deviation in each computation. We attempt to discuss how this comes about in the discussion section below. The best approximation for $\rho_{max} = 40$ was for $N = 379$ with a mean deviation of ≈ 0.135.

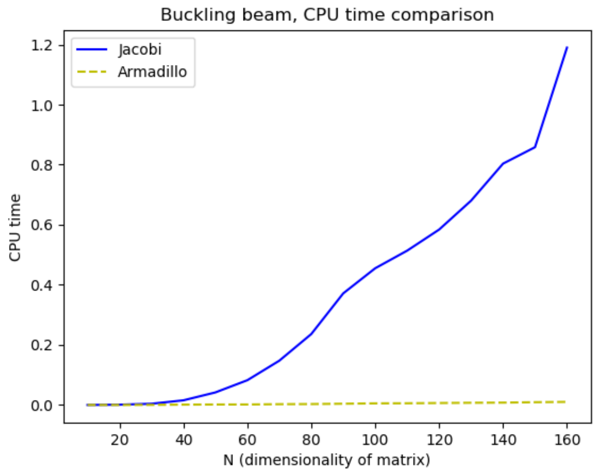

Figure 4. Comparison of CPU time usage between the jacobi method and armadillo's eigsym function.

IV. DISCUSSION

In table I we can compare the computation time between the different eigenvalue solvers. The Jacobi method for a larger matrix is much slower than for the other two methods. This is due to the fact that the Jacobi method does not take advantage of the fact that the matrix is tridiagonal, but goes through all the matrix elements of the matrix. Some of the matrix elements that it sets to zero, can additionally become non-zero in a later rotation [1]. This makes the method unpractical to use for larger matrices, even though it is direct and intuitive in nature.

The bisection method is considerably faster, as this method is based on the fact that our matrix is tridiagonal. In addition, even though we use the method to find all the eigenvalues here, we can use it to only find some eigenvalues if we are not interested in all, thereby saving time. However, the bisection method does not return the eigenvectors of our matrix, so if we are interested in these as well we need a different method.
The armadillo function `eig_sym` returns both eigenvalues and eigenvectors, and is around the same speed (a little faster) as the bisection method for $N = 100$. However, it might not be as fast as the bisection method for larger N, or if we are only interested in some eigenvalues.

In figure 2 we saw how the number of similarity transformations increased for increasing values of N. The relationship looks close to that of a function going as n^2 but with some constant term. Analytically we do expect this relationship, the jacobi method uses around $3n$ to $5n$ rotations and for each we need to identify the new maximum element in the matrix which takes some constant times n more operations. The rest of the results for the buckling beam using the jacobi method seems somwhat straight forward. We have mostly tested the validity of our method against armadillo's `eig_sym` function and seen that the results corresponds well between the two methods.
The results of Schroedinger's equation for one electron seems to be physically valid. The eigenvector shown in the plot should represent the probability density of the electron in its ground state. Since we only look at the radial part this shows us the probability of finding the electron at a given (rescaled) position ρ outside the proton. If we then think about this in three dimensions it gives us the famous "electron cloud" surrounding the proton. Another thing to notice is that we have set the angular quantum number l to 0 and therefore this cloud is uniformly distributed around the proton at the center.

In our results we also noticed that the armadillo eigsym function returned eigenvalues corresponding to those of the jacobi method, but the eigenvector was for some reason returned negative. This would not be a problem in a quantum mechanical problem since when computing the actual probability we take the absolute value and square the inner product of the vectors (states).

The results obtained by finding the relative error be-

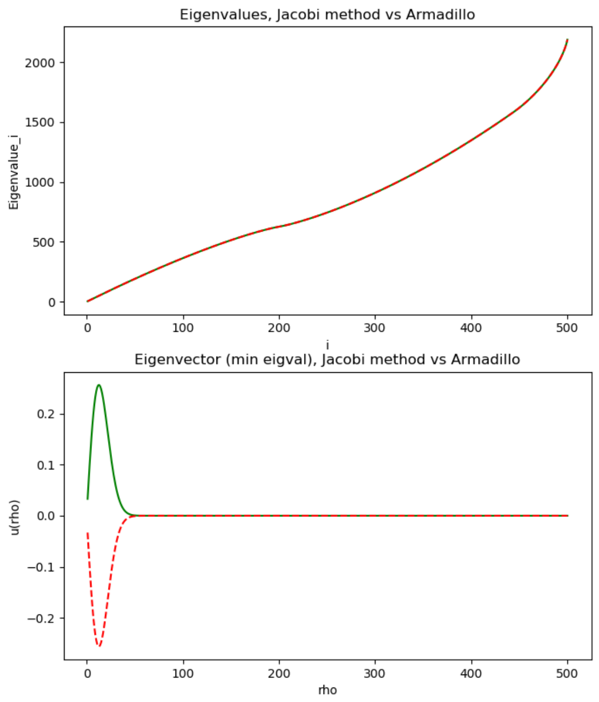

Figure 5. Comparison of eigenvalues (energies) and the eigenvector for the lowest eigenvalue using the jacobi method (green) and armadillo's eigsym (dashed red).

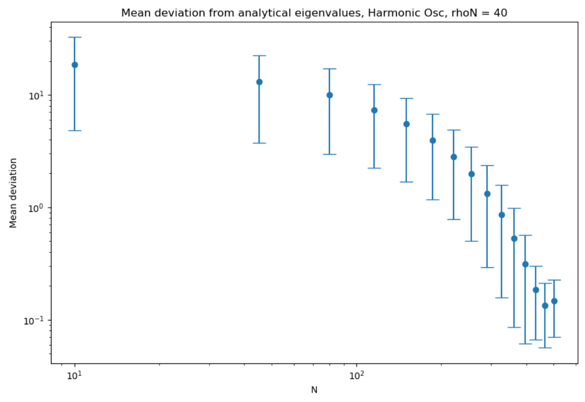

Figure 6. Logarithmic plot showing the mean deviation in the relative error between numerically obtained eigenvalues and analytical eigenvalues. The bars represent the standard deviation.

tween the analytical eigenvalues and those obtained with the jacobi method is somewhat hard to interpret. It is clear that both our choice of N and ρ_{max} are not independent of each other, hence, it seems that increasing both as much as possible while still keeping ρ_{max} about a factor of ten smaller than N is a good approach. But we were not able to find any analytic relationship which would give us the perfect choice of ρ_{max} for each choice of N. This might not even be possible. Our program was able to show that for increasing N and a large value of ρ_{max} the mean deviation in the relative error did drop substantially. However, it seems that it never got close to averaging within 4 leading digits after the decimal point in precision. We could have set up an algorithm for finding the best possible value of ρ_{max} for a large value of N or even better made a matrix of differing values of N and ρ_{max} and found the minimum of this. Perhaps this type of exercise would be better suited for machine learning algorithms.

V. CONCLUSION

In this project we have looked at the eigenvalue solutions of differential equations by rewriting the equations on matrix form using the differential matrix operator. We also rescaled the problems to simplify the solutions. To solve the eigenvalue problems we implemented the jacobi method and the bisection method and compared the two. The first problem we looked at was for the buckling beam. After setting up the symmetric matrix for the problem we compared the analytic eigenvalues with those obtained with the Armadillo function `eig_sym`, shown in figure 1. We also plotted the number of similarity transformations as a function of N and the CPU time as a function of N, shown in figure 2, which corresponded to the expected analytical results. To check the validity of our implementation of Jacobi's method we compared the eigenvalues and the eigenvector corresponding to the lowest eigenvalue to those obtained using `eig_sym`, as shown in figure 3. Lastly we compared the CPU time with $N = 10$ and 100 for Jacobi's method, `eig_sym` and the Bisection method, the results are presented in table I.

The other problem we tackled were that of an electron in a three dimensional harmonic oscillator potential which we solved using Schroedinger's equation. To solve it we discretized the differential operator and scaled the equation. We then used our implementation of Jacobi's method to solve the problem and find the eigenvalues and corresponding eigenvectors, where the eigenvalues corresponds to the energies and the eigenvectors to the energy eigenstates. The resulting plots can be found in figure 5. We also attempted to find the value of N which approximated the problem with an uncertainty of $\sim 10^{-4}$ but was unable to achieve this. The mean deviation obtained is found in figure 6.

This project has served as a good example of how useful scaling of equations is for computational science and also through the problems shown how to discretize equations. We also learned some useful numerical algorithms and got a much better feeling for how to structure the code and organize it into units which can be tested to make sure the code works as intended.

REFERENCES

[1] Morten Hjort-Jensen. "Computational Physics: Lecture Notes Fall 2015". 2015.

[2] Tom Lyche. "Lecture Notes for Mat-inf 4130". In: The name of the publisher, June 2017. Chap. 1.2-1.3, pp. 43–50.

[3] R. S. Martin W. Barth and J. H. Wilkinson. "Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection". In: *Numerische Mathematik* 9 (1967), pp. 386–393.