Calculating eigenvalues using Jacobi's rotational algorithm

Anders P. Åsbø
(Dated: October 1, 2019)

This paper focused on the implementation, and application of Jacobi's rotational algorithm to find the eigenvalues of Tridiagonal matrices numerically. The physical problems considered were a buckeling beam, and both a single electron and two electrons trapped in a harmonic oscillator potential.

The goal is to evaluate the usability of the implementation, and look at possible improvements that can be made in future implementations of Jacobi's rotational algorithm.

The implementation of Jacobi's algorithm looked at in this paper, was able to calulate the eigenvalues of the buckling beam problem with high accuracy (table II), but could only reproduce the eigenvalues of the single-electron problem within 2 leading decimals.

I was able to reproduce one of the results from Taut, M. (1993)[1]. However, the reliability of this result is suspect at best.

Overall, my implementation of Jacobi's algorithm is not particularly resource efficient. There are several improvements that can be made in future implementations. It can be used somewhat reliably for problems with a finite maximum variable, for $N \leq 200$, but require to be run on better hardware to allow higher N when necessary.

CONTENTS

I. INTRODUCTION

The focus of this paper was the implementation, and application of Jacobi's rotational algorithm to find the eigenvalues of Tridiagonal matrices numerically. Reliably finding eigenvalues is a crucial part of many scientific and mathematical disciplines. The physical problems considered were a buckeling beam, and both a single electron and two electrons trapped in a harmonic oscillator potential.

The goal is to evaluate the usability of the implementation, and look at possible improvements that can be made in future implementations of Jacobi's rotational algorithm.

II. FORMALISM

A. The buckling beam problem

The pretense for implementing Jacobi's algorithm is the classical problem of a beam of length L fastened in both ends $x_0 = 0$, $x_L = L$. The beam is allowed to be displaced in the y-direction with displacement $u(x)$, while $u(0) = u(L) = 0$. The displacement is driven by a force F at $(L, 0)$ towards the origin. The displacement is then described by

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

where γ is a constant dependent on the physical properties of the beam[2].

By scaling the differential equation with $\rho = \frac{x}{L}$, such that $\rho \in [0, 1]$, and introducing the parameter $\lambda = FL^2/\gamma$[2], the differential equation becomes

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

Finally, the equation can be discretized with

$$u'' \approx \frac{u(\rho+h)-2u(\rho)+u(\rho-h)}{h^2},$$

where $h = \frac{\rho_N-\rho_0}{N}$ with N steps[2]. The resulting discretization becomes

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

which can be written as the matrix equation

$$A\vec{u}=\lambda\vec{u},$$

where A is an $(N-2)\times(N-2)$ tridiagonal Toeplitz matrix with the diagonal elements $d = 2/h^2$, and upper and lower diagonals with elements $a = -1/h^2$. This matrix happens to have analytical eigenvalues

$$\lambda_j = d+2a\cos\left(\frac{j\pi}{N+1}\right),$$

where $j = 1, 2, ..., N$[2].

B. The Jacobi rotational algorithm

The goal of the Jacobi rotational algorithm is to reduce a matrix A to a diagonal matrix B where the elements along the diagonal are the eigenvalues λ_i of A. This is usually done by finding a matrix S such that

$$B = S^T AS,$$

and $S^T = S^{-1}$[3]. The Jacobi algorithm achieves this by choosing the elements of S to be equal to the coresponding identity matrix, except for the elements $s_{kk}, s_{ll} = \cos\theta$ and $s_{kl} = \pm\sin\theta$, $s_{lk} = -s_{kl}$[3], and applying the $B = S^TAS$ transformation repeatedly, until the non-diagonal elements of B are sufficiently close to zero. Doing this we get a system of equations for the various elements of the resulting matrix B

$$b_{ii} = a_{ii} i \neq k, i \neq l,$$

$$b_{ik} = a_{ik}c - a_{il}s, i \neq k, i \neq l,$$

$$b_{il} = a_{il}c + a_{ik}s, i \neq k, i \neq l,$$

$$b_{kk} = a_{kk}c^2 - 2a_{kl}cs - a_{ll}s^2,$$

$$b_{ll} = a_{ll}c^2 - 2a_{kl}cs - a_{kk}s^2,$$

$$b_{kl} = (a_{kk}-a_{ll})\,cs - a_{kl}\left(c^2-s^2\right),$$

$$b_{lk} = -b_{kl},$$

where $c = \cos\theta$, $s = \sin\theta$, and k, l are chosen such that a_{kl} is the non-diagonal element in A with the largest absolute value.

To chose an rotational angle θ, the quantities $\tan\theta = t = s/c$ are defined such that

$$\cot 2\theta = \tau = \frac{a_{ll}-a_{kk}}{2a_{kl}},$$

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

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

$$s = ct,$$

[3].

C. Eigenvalues of a one-electron Hamiltonian

To test the implementation of Jacobi's algorithm, as outlined in section II B, a quantum mechanical system consisting of an electron trapped in a radialy symetric harmonic oscillator potential $V(r) = \frac{1}{2}mkr^2$ was chosen as a test case. the corresponding radial equation is

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

where $\hbar$ is Planck's constant, m is the electron mass, $R(r)$ is the radial wavefunction, and $l = 0, 1, 2, 3, ...$ is the orbital momentum of the electron[2]. By substituting in $R(r) = (1/r)u(r)$, and introducing the scaled variable $\rho = (1/\alpha)r$, where α is some constant, the equation can be reduced to

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

where α is chosen such that $\frac{mk}{\hbar^2}\alpha^4 = 1$, and

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

[2].

After discretization as in section II A, the problem results in the system of linear equations

$$-\frac{1}{h^2}u_{i+1}+\left(\frac{2}{h^2}+\rho^2\right)u_i-\frac{1}{h^2}u_{i-1}=\lambda u_i,$$

with the requirement that $u(0) = u(\infty) = 0$[2]. This can be written as a matrix equation where the matrix A is a tridiagonal matrix with $d = \left(\frac{2}{h^2}+\rho^2\right)$ along the diagonal, and $e = -\frac{1}{h^2}$ as the non-diagonal elements. The step-size h is as defined in section II A.

This specific problem has the analytical eigenvalues $\lambda = 3, 7, 11, 15$[2].

D. Eigenvalues of a two-electron Hamiltonian

The radial equation for a non-interacting electron pair, can be written as the product of two single-electron radial equations

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

where $r = r_1 - r_2$ is the difference between the radial coordinates of the two electrons, and $R = \frac{1}{2}(r_1 + r_2)$ is the coordinate of the center of mass[2].

By seperating the radial wavefunction $u(r,R) = \psi(r)\phi(R)$, and adding in the repulsive Coloumb interaction between the electrons $V(r) = \beta e^2/r$, where $\beta$is a konstant and e is the electron charge, the r-dependent Scroedinger equation becomes

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \omega_r^2\rho^2\psi(\rho) + \frac{1}{\rho} = \lambda\psi(\rho),$$

where $\rho \in [0,\infty)$ is the same dimensionless variable from section II C, the frequency $\omega_r = \frac{1}{4}\frac{mk}{\hbar^2}\alpha^4$, α is chosen such that $\frac{m\alpha\beta e^2}{\hbar^2} = 1$, and $\lambda = \frac{m\alpha^2}{\hbar^2}E$[2].

Doing the same discretization as in section II A and section II C results in the linear equations

$$-\frac{1}{h^2}u_{i+1} + \left(\frac{2}{h^2} + \omega_r^2\rho^2 + \frac{1}{\rho}\right)u_i - \frac{1}{h^2}u_{i-1} = \lambda u_i,$$

and a tridiagonal matrix A with diagonal elements $d = \left(\frac{2}{h^2} + \omega_r^2\rho^2 + \frac{1}{\rho}\right)$ and non-diagonal elements $e = -\frac{1}{h^2}$. The step-size h is defined as in section II A.

This wave equation has analytical eigenvalues for certain ω_r as described in Taut, M.'s paper in Physical Review (1993) [1].

III. IMPLEMENTATION

A. Constructing the matrix and testing agianst the NumPy solver

As a prequel to implementing the jacobi algorithm, "program.py" was written to construct the tridiagonal Toeplitz matrix, and find its eigenvalues using the numpy.linalg.eig() method provided by the NumPy package. "program.py" also includes a test comparing the numerically found eigenvalues with the expected results from the analytical expression in section II B, with a tolerance of 10^{-10}.

B. Implementing Jacobi's rotational algorithm

Jacobi's algorithm was implemented in "jacobi.py". The afformentioned program takes in a value N, and sets up the parameters as well as creating the Toeplitz matrix using the function from "program.py". A mask is created and used when calculating the norm of the non-diagonal elements in the matrix. The jacobi function is called, returning the diagonal elements.

The jacobi function consists of a while-loop that runs until the norm of the non-diagonal elements are less than a tolerance of 10^{-20}. The loops starts by finding a_{kl} with the maximum absolute value, then uses the returned indices to retrive the a_{kk}, and a_{ll}. The parameters τ, t, c, s are all calculated, with t being chosen such that $|\theta| \leq \frac{\pi}{4}$ to ensure minimal difference between A and B[3].

Furthermore, a for-loop is started that runs through A and calculates the non-diagonal elements except for $b_{kl}, b_{lk}, b_{ll}, b_{kk}$. After the for-loop finishes, the remaining matrix elements are calculated, as well as the new norm of the non-diagonal elements, and the while-loop has completed one full loop.

When the norm of the non-diagonal elements is below the set tolerance, the while-loop finishes, and the matrix B is returned by the function.

The find_max function in "jacobi.py" uses two nested for-loops to run over every non-diagonal element in A and check if the current element has a larger absolute value than the previously stored a_{kl}, with the starting value being $a_{kl} = 0$. Once a valid value is found, it is stored and used to evaluate the remaining elements until it is replaced with a new maximum, or the loops end. The new a_{kl} is then returned together with the indices k, l.

Finally, the numerical eigenvalues are extracted, sorted and printed to the terminal.

The program files also includes "unittest.py", which tests the corespondence between the numerical eigenvalues found using the jacobi function on the tridiagonal matrix from section II A, and the analytical eigenvalues, with a tolerance of 10^{-10}. A second unit test in "unittest.py" tests if the find_max function is able to always pick out the element with the largest absolute value in a randomized 5×5-matrix.

C. Implementing the single-electron problem

To solve the eigenvalues of the problem outlined in section II C, "oneelectron.py" was written. The program takes in the number of grid points. Ten it sets the maximum value $\rho_N \approx \infty$ of ρ, which after trial and error was determined to be 12.5, with the computational resources I had available.

The step-size, and matrix elements are calculated, and the matrix is created, as well as a mask that conceals the diagonal when evaluating the norm of A. The matrix is diagonalized using the jacobi function from "jacobi.py", and the eigenvalues are retrieved.

Finally, the program prints the eigenvalues, as well as their relative errors.

D. Implementing the two-electron problem

The implementation of the problem outlined in section II D is almost identical to the implementation in section III C, and can be found in "twoelectron.py". The difference is in the diagonal elements of A, and that "twoelectron.py" does not calculate the relative error of the eigenvalues.

IV. ANALYSIS

A. Results from testing the NumPy solver on the buckeling beam problem

Numerical eigenvalue	Analytical eigenvalue
6.69872981	6.69872981
25	25
50	50
75	75
93.30127019	93.30127019

Table I. Numerical and analytical eigenvalues of the tridiagonal Toeplitz matrix for the buckling beam problem, using the NumPy solver. $N = 5$

Table I shows the resulting numerical, and analytical eigenvalues of "program.py" being run with $N = 5$. The NumPy solver could handle up to and including $N = 79$, before exceeding the tolerance of 10^{-10} with a maximum error of $1.382\,432 \times 10^{-10}$. With $N = 10^3$, which was the highest tested, the maximum error was only $4.097\,819 \times 10^{-8}$.

B. Results of implementing the buckling beam problem using Jacobi's algorithm

Numerical eigenvalue	Analytical eigenvalue
6.69872981	6.69872981
25	25
50	50
75	75
93.30127019	93.30127019

Table II. Numerical and analytical eigenvalues of the tridiagonal Toeplitz matrix for the buckling beam problem, using the jacobi.py solver. $N = 5$

Table II shows the resulting numerical, and analytical eigenvalues of "jacobi.py" being run with $N = 5$. The jacobi.py solver could handle up to and including $N = 5$, before exceeding the tolerance of 10^{-10} with a maximum error of $1.145\,963 \times 10^{-10}$. With $N = 200$, which was the highest tested, the maximum error was $5.587\,935 \times 10^{-9}$.

C. Results of the single-electron problem with Jacobi's algorithm

Numerical eigenvalue	Analytical eigenvalue	Relative error
3.01384183	3	0.00461394
7.02900535	7	0.00414362
11.04021305	11	0.00365573
15.04745905	15	0.00316394

Table III. Numerical and analytical eigenvalues of the Hamiltonian for the single electron problem, using the jacobi.py solver. $N = 200$, $\rho_N = 12.5$

Numerical eigenvalue	Analytical eigenvalue	Relative error
3.04073901	3	0.01357967
7.0396407	7	0.00566296
10.97051694	11	0.00268028
14.83148019	15	0.01123465

Table IV. Numerical and analytical eigenvalues of the Hamiltonian for the single electron problem, using the jacobi.py solver. $N = 50$, $\rho_N = 12.5$

Numerical eigenvalue	Analytical eigenvalue	Relative error
3.01476709	3	0.00492236
7.03363428	7	0.0048049
11.05151444	11	0.00468313
15.06840746	15	0.0045605

Table V. Numerical and analytical eigenvalues of the Hamiltonian for the single electron problem, using the jacobi.py solver. $N = 200$, $\rho_N = 6.25$

Numerical eigenvalue	Analytical eigenvalue	Relative error
3.05613118	3	0.01871039
7.11734334	7	0.01676333
11.16211274	11	0.01473752
15.190338	15	0.0126892

Table VI. Numerical and analytical eigenvalues of the Hamiltonian for the single electron problem, using the jacobi.py solver. $N = 50$, $\rho_N = 6.25$

By comparing table III, table IV, table V, and table VI, it appears that adjusting N has the most noticable effect on the accuracy of the jacobi.py solver. I was unable to push the accuracy further, due to the limited single-core performance of the hardware that was used.

D. Results of the single-electron problem with Jacobi's algorithm

Numerical eigenvalue	ω_r
0.16645674	0.01
1.2501149	0.25
0.16645674	0.5
4.07545554	1.0

Table VII. Numerical eigenvalues of the Hamiltonian for the two-electron problem, using the jacobi.py solver. $N = 100$, $\rho_N = 15.794$

Table VII shows the lowest eigenvalues of the two-electron problem outlined in section II D calculated using the jacobi.py solver. The values of N, and ρ_N was chosen by trying to make the value for $\omega_r = 0.25$ match the value of $0.5\lambda = 0.625$ found by Taut, M. (1993) (see Table I for $n = 4$ in [1]), which is not the lowest eigenvalue. However, this was not a reliable way of adjusting parameters, as a slight nudge gave significantly different results. Perhaps if the implementation of Jacobi's algorithm was better optimized, and/or the hardware running the calculation was better, I could have achieved more reliable results.

The remaining values in table VII shows the lowest eigenvalue, and presumably the ground state energy.

E. Discussion of implementation

Overall, my implementation of Jacobi's algorithm is not particularly resource efficient. Possible improvements include looking for terms, and factors that can be precalculated to reduce the number of FLOPS, as well as other possible ways to store the matrix than as a $2D$-NumPy array, seeing as this stores a lot of 0-elements in memory that will never be used in calculations.

Furthermore, rewriting the jacobi function to make it compatible with the Numba jit-compiler's nopyhton-mode, so as to take full advantae of the jit-compiler. Alternativly, implimenting the algorithm in a lower-level programming language than Python 3.6, such as c++, would likely speed up the calculations. I would also look at the way the step-size is implemented, as I am unsure if I have understood the definition correctly.

Overall, the implementation seems usable for problems that have a finite max value for their function variable ρ_N. While implimentations that require a sufficently large $\rho_N \approx \infty$, would require hardware with better computing recourses, such that a matching sufficiently large N can be used. Otherwise it is hard to get reliable results with high accuracy.

V. CONCLUSION

The implementation of Jacobi's algorithm looked at in this paper, was able to calulate the eigenvalues of the buckling beam problem with high accuracy as seen in table II, but could only reproduce the eigenvalues of the single-electron problem within 2 leading decimals.

Furthermore, the implementation could reproduce one of the results from Taut, M. (1993)[1] to 3 leading decimals. However, the reliability of this result is suspect at best.

Overall, my implementation of Jacobi's algorithm is not particularly resource efficient. There are several improvements that can be made in future implementations. The implementation discussed in this paper can be used somewhat reliably for problems with a finite maximum variable, for $N \leq 200$, but require to be run on better hardware to allow higher N if the problem beeing solved requires the maximum variable to approximate infinity.

[1] M. Taut, Phys. Rev. A **48**, 3561 (1993).
[2] Department of Physics, *Project 2 - Computational Physics I FYS3150/FYS4150*, Tech. Rep. (2019).
[3] M. Hjorth-Jensen, *Computational Physics Lectures: Eigenvalue problems*, Tech. Rep. (2019).

Appendix A: Program files

All code for this report was written in Python 3.6, and the complete set of program files can be found at https://github.com/FunkMarvel/CompPhys-Project-2.

1. program.py

https://github.com/FunkMarvel/CompPhys-Project-2/blob/master/program.py

2. jacobi.py

https://github.com/FunkMarvel/CompPhys-Project-2/blob/master/jacobi.py

3. oneelectron.py

https://github.com/FunkMarvel/CompPhys-Project-2/blob/master/oneelectron.py

4. twoelectron.py

https://github.com/FunkMarvel/CompPhys-Project-2/blob/master/twoelectron.py

5. unittest.py

https://github.com/FunkMarvel/CompPhys-Project-2/blob/master/unittests.py