Project 2 in FYS3150

Bendik Steinsvåg Dalen, Ulrik Seip

October 3, 2018

1 ABSTRACT

In this project we have implemented Jacobian algorithm, to find eigenvectors, and their corresponding eigenvalues in tridiagonal matrices. We then used this to model a harmonic oscillator problem in three dimensions, with one and two electrons. This turned out to be a computationally heavy, but relatively accurate method.

2 INTRODUCTION

Finding eigenvectors analytically is complicated, and can be tedious, and this is why it is much more convenient to do so numerically. A common way of doing this is by the application of the Jacobian method. Essentially we rotate one matrix element at a time, always taking the one with the highest absolute value, until all but the diagonal elements are essentially zero. All transformations are also applied to an identity matrix that then turns into our eigenvectors.

3 METHOD

3.a Implementing the Jacobian algorithm

The implementation follows a standard recipe: We start with the relation

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

This can be used to find the angle θ that makes the non-diagonal matrix elements of the transformed matrix $a_{kl} = 0$. The quadratic equation is obtained using $\cot 2\theta = 1/2(\cot\theta - \tan\theta)$.

$$t^2 + 2\tau t - 1 = 0 \tag{1}$$

Which gives us

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

c and s are then obtained by

$$c = \frac{1}{\sqrt{1 + t^2}} \tag{3}$$

Which gives us

$$t = -\tau \pm \sqrt{1 + \tau^2} \tag{4}$$

c and s are then obtained by

$$c = \frac{1}{\sqrt{1 + t^2}} \tag{5}$$

and

$$s = tc \tag{6}$$

We then use the rotational factors c and s to rotate every other element in the matrix according to their position, giving us a new diagonal, that is slightly closer to the eigenvalues, and new matrix elements elsewhere, slightly closer to 0. The same is done for a matrix that tarts out as an identity matrix, with the purpose of transforming it into the original matrix' eigenvectors. This is all done in the programming language Julia for high efficiency. See the documentation in section 6.a for further explanation.

3.b Testing the code

For testing the algorithm we have implemented two tests. One for checking if the largest element in the matrix is correctly located, and one for testing if the resulting eigenvalues are correct. The first one is more useful for development purposes, whilst the second one is essential for validating that our implementation works correctly. See section 6.b for the test functions.

3.c Quantum dots in three dimensions, one electron

Now that we had a general algorithm we used it to model a electron that moves in a three-dimensional harmonic oscillator potential. In other words, we looked for the solution of the radial part of Schroedinger's equation for one electron, which reads

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r). \tag{7}$$

This problem also has analytical solutions, so we can test how accurate our algorithm is.

We decided to limit our experiment to the ground state of $l=0$. After introducing the constant $\alpha=\left(\frac{\hbar^2}{mk}\right)^{1/4}$, and the variables $\lambda=\frac{2m\alpha^2}{\hbar^2}E$ and $\rho=\frac{r}{\alpha}$, the Schroedinger's equation becomes

$$-\frac{d^2}{d\rho^2}u(\rho)+\rho^2u(\rho)=\lambda u(\rho). \tag{8}$$

See section 6.c for futher details.

Since we are working in radial coordinates we have $\rho\in[0,\infty)$. Since we can't represent infinity on a computer we have to find an aproximation, which we will come back to later. For now we define $\rho_{min}=10^{-6}$ and ρ_{max} to represent the minimum and maximum values of ρ. We used 10^{-6} instead of 0 to avoid any potential divisions by zero.

Function 8 is an differential equation that can be modeled similarly to earlier. If we have N mesh points we get a step length

$$h=\frac{\rho_{max}-\rho_{min}}{N}. \tag{9}$$

The value of ρ at a point i is then

$$\rho_i=\rho_0+ih \qquad i=1,2,\dots,N. \tag{10}$$

We can rewrite the Schroedinger equation for a value ρ_i as

$$-\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}+\rho_i^2u_i=\lambda u_i). \tag{11}$$

We then introduced

$$d_i=\frac{2}{h^2}+\rho_i^2, \tag{12}$$

and

$$e_i=-\frac{1}{\hbar^2}, \tag{13}$$

which gives us

$$d_iu_i+e_{i-1}u_{i-1}+e_{i+1}u_{i+1}=\lambda u_i. \tag{14}$$

We then wrote the latter equation as a matrix eigenvalue problem

$$\begin{bmatrix} d_0 & e_0 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_1 & e_1 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_2 & e_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots e_{N-1} & d_{N-1} & e_{N-1} \\ 0 & \dots & \dots & \dots & \dots & e_N & d_N \end{bmatrix}\begin{bmatrix} u_0 \\ u_1 \\ \dots \\ \dots \\ \dots \\ u_N \end{bmatrix}=\lambda\begin{bmatrix} u_0 \\ u_1 \\ \dots \\ \dots \\ \dots \\ u_N \end{bmatrix}. \tag{15}$$

This is effectively the same problem as earlier, only that we have to add ρ_i^2 to the diagonal elements.

To solve the problem we implemented the matrix in Julia, and ran it through the algorithm in section 6.a, with a tolerance of 10^{-4}. The implemtation of the matrix can be seen in section 6.d. What we then did was test different approximations for ρ_{max} and N to find some values that are accurate and don't require a extemly large N. Our goal was to reproduce the analytical eigenvalues with four leading digits after the decimal point. We looked both at the accuracy of the first eigenvalue, and used the Julia function *Statistics.std*() on the difference between our numerical apppoximation and the analytical solutions, to get an idea of how close the results were overall. *std*() is an function that finds the standar diviation for an array.

3.d Quantum dots in three dimensions, two electrons

4 RESULTS

4.a The implementation

We see a fairly linear correlation between the number of matrix elements n^2 and the required amount similarity transformations. The computation time for each matrix was also proportional to n^2 In figure 1 the tolerance for deviation from 0 in the non diagonal elements was $1e-4$. We found this to be the best balance between accuracy and efficiency.

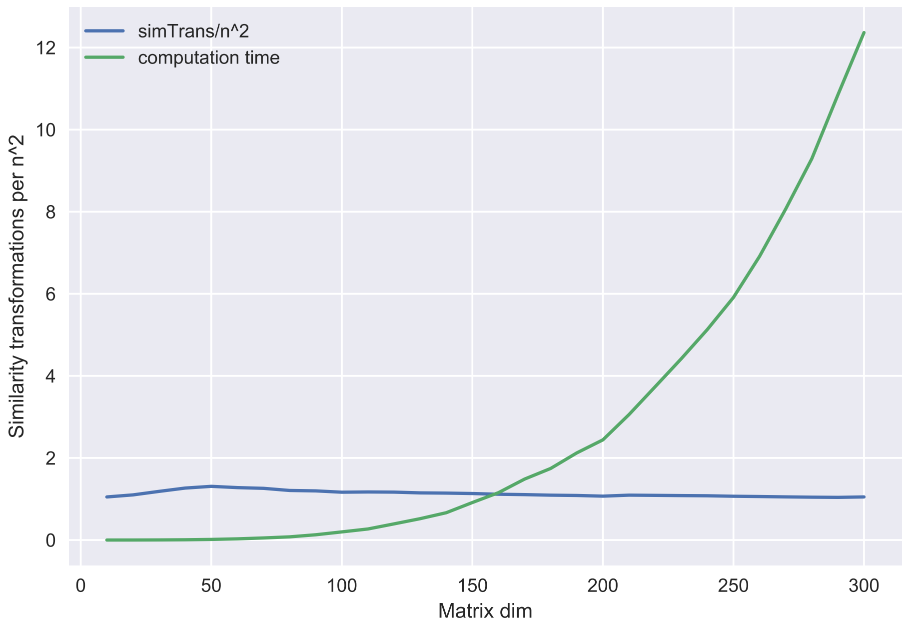

Figure 1: A plot of the required number of rotations as a function of n: $rotations/n^2$, and the time used for calculating the eigenvalues of a $n \cdot n$ matrix.

5 CONCLUSIONS

With an accuracy of more digits than our editor cared to print out the Jacobi method, with a tolerance for non diagonal values of up to $1e-4$, seems to be an efficient and precise algorithm for computing eigenvectors and eigenvalues. The computation time is proportional to the number of matrix elements, and so the realistic limit for matrix size should be around $10^4 \cdot 10^4$. With $300 \cdot 300$ taking 12 seconds, $10^4 \cdot 10^4$ should take about

$$\frac{(10^4)^2}{300^2} \cdot 12s \approx 4h \tag{16}$$

on a normal laptop. On a supercomputer this would of course be different, and i presume our implementation could have been further vectorised for greater efficiency.

6 APENDICES

6.a Integration loop from rotator.jl

```
            eigvals[i] = M[i, i]
        end
    return sort(eigvals)
end

#matrix printing function lendt form Lars W. Dreyer and Gabriel S. Cabrera
function print_matrix(M)
    N = size(M)[1]
    for i=1:N
```

```
        for j=1:N
            print("$(@sprintf("%7.2f ", M[i,j]))")
        end
        println()
    end
    println()
end

function off(a)
    return maximum(a^2) #this might work? p. 217 lecture notes.
end

function maxKnotL(a)  #finds larges element that is not on the diagonal on matrix a
    max = 0
    kl = [1,1]
    n = Int64(length(a[1,:]))
    for l = 1:n
        for k = l+1:n
            ma = abs(a[k, l])
            if (ma > max)
                max = ma
                kl = [k, l]
            end
        end
    end
    return kl[1], kl[2]                #k, l
end

function rotate(a, tol)          #den faktiske rotasjonslokken. Tar inn en matrise a og noyaktighet.
    n = Int64(length(a[1,:]))     #initiates n for later use
    r = Matrix{Float64}(I, n, n)  #initialising eigenvector matrix
    counter = 0                   #teller antall "similarity transformaitons"
    k, l = maxKnotL(a)            #finds indices of matrix element with highest value
    countermax = n^3
    while (abs(a[k ,l]) > tol) && (counter < countermax)  #this is the actual loop
        rotation(a, r, k, l, n)
        counter += 1
        k, l = maxKnotL(a)
    end
    return a, r, n, counter, tol     #a = newA, r = egenvektorer, n = dim(a), counter = n(simTrans) og tol = max value
end


function rotation(a, r, k, l, n)
    if (a[k, l] != 0.0)
        #kl = maxKnotL(a)
        tau = (a[l, l] - a[k, k])/(2*a[k, l]) #blir ikke dette alltid null?
        if (tau > 0)
            t = -tau + sqrt(1.0 + tau^2)
        else
            t = -tau - sqrt(1.0 + tau^2)
        end
        c = 1/sqrt(1.0 + t^2)
        s = c*t
    #end
```

6.b Test functions

```
function test_maxKnotL()
    test_matrix = ones(Float64, 10, 10)
    test_matrix[1, 4] = 3
    test_matrix[4, 1] = 3
    k, l = maxKnotL(test_matrix)
    if test_matrix[k, l] < 3
        error("maxKnotL returns wrong value")
    end
end
```

```
function test_eigenvalues()
    #creating testable values
    n = Int64(5)
    a = zeros(Float64, n, n)
    for i in range(1, step = 1, length = n)
        for j in range(1, step = 1, length = n)
            if ((j == i+1) || (j == i-1))
                a[i, j] = 2                   # b = 2
            end
            if (i == j)
                a[i, j] = 5                   # a = 5
            end
        end
    end
    test_values = [5-2*sqrt(3), 3, 5, 7, 5+2*sqrt(3)]
    #calculating values
    a_, thing1, thing2, thing3, thing4 = rotate(a, 1e-5)
    tol = 1e-3
    #sorting values into usable format
    values = zeros(5)
    for i = 1:5
        values[i] = a_[i,i]
    end
    values = sort(values)
    #tests each element
    for i = n:5
        if abs(values[i]-test_values[i]) > tol
            error("Eigentest failed: expected: $test_values, got $values")
        end
    end
end

test_eigenvalues()
```

6.c Math for Quantum dots in three dimensions, one electron

We begin with

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r). \tag{17}$$

Firstly we had that $l=0$. We then substitued $R(r)=u(r)/r$, and got

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+V(r)u(r)=Eu(r). \tag{18}$$

In our case $V(r)$ is equal to $(1/2)kr^2$. We now introduce a dimensionless vaiable, $\rho = r/\alpha$

6.d Implementing a matrix in Julia

```
#function for making the matrix
function make_mat(rho_minn, rho_max, n)
    h = (rho_max-rho_minn)/n #the number of steps
    e = -1/h^2 #the non-diagonal matrix element

    d = zeros(n) #array to hold the diagonal matrix element
    #finds the diagonal matrix element for each rho
    for r=1:n
        d[r] = 2/h^2 + (rho_minn + r*h)^2
    end

    a = zeros(Float64, n, n) #the matrix, we now have to fill in the diagonal values
    a[1,1] = 2/h^2 + rho_minn^2 #the first elements
    a[1,1+1] = e
    for i=2:n-1 #all the other elements
        a[i,i-1] = e
        a[i,i] = d[i]
        a[i,i+1] = e
```

```
    end
    a[n,n-1] = e #the last elements
    a[n,n] = 2/h^2 + rho_max^2
    return a
end
```

7 REFERENCES

References

[1] Computational Physics, Lecture Notes Fall 2015, Morten Hjort-Jensen p.215-220