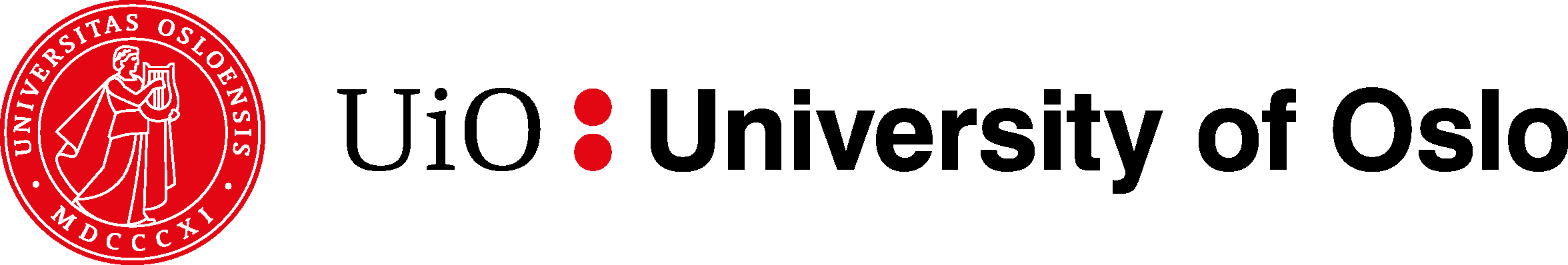
Computational Physics

Project 2



Hunter Wilhelm Phillips (*hunterp*)

October 3, 2018

GitHub Repository at <https://github.com/robolux/Computational_Physics>

**Abstract**

This project explores the numerical solving of a second order differential equation in terms of a matrix. The use of general gaussian elimination, specialized tridiagonal solver, and LU decomposition allows for varying approaches to solving the problem. An error analysis was also performed to determine the costs of increased computational time with respect to a decrease in error. To verify the correlated computation time, a timing sequence was executed to compare with the recorded error analysis. The results show that the specialized tridiagonal algorithm has the optimal attributes of low computation time and low error.

1. Discussion of Problem Statement

In this project, the goal is to analyze a 3-dimensional harmonic oscillator potential with spherical symmetry that acts on a system of electrons. It is assumed that there are no angular components for the particles, only radially dependent movement. The two particles interact with respect to Coulomb’s force acting upon the bodies.

1.1 One Particle System

The first step to explaining a system containing one particle is to look at the radial Schrödinger’s equation (1):

(1)

where is the time independent wave function defined as the

spatial second order derivative written in spherical coordinates

is the harmonic oscillator potential

is the energy of the harmonic oscillator in three dimensions

By taking this system and displaying the closed form for the energy that are eigenstates of the above wave function.

Using n = 0, 1, 2 …. until an endpoint is desired the simplification of (1) can be simplified with the discretization of with the coupled boundary conditions of and . This results in (2) being formed:

(2)

The introduction of a dimensionless variable where the constant has a dimensional length given forms (3):

(3)

To make the potential coefficient equal to 1 the constant value must be fixed with the decision resulting in the following:

This allows the grounding equation to become dimensionless, however resulted in a change of the energy eigenvalues in the original wave function. By defining the following, we can subsequently rewrite Schrödinger’s equation:

(4)

This final equation (4) is the derived equation to solve numerically and the associated three-dimensional closed form for the eigenvalues, , can be coupled for a solution index.

1.2 Two Particle Solution

The second step is the explanation of two electron system in a harmonic oscillator well that interact with each other with a repulsive Coulomb interaction. By considering no repulsive Coulomb interactions, the following Schrödinger’s equation in spherical coordinates can be derived:

(5)

By separating the two-electron energy from the wave function in (5) through the consideration of the relative coordinates and the coupled center-of-mass coordinate

the coordinate shifted Schrödinger’s equation has become:

(6)

By adding the Coulomb’s potential (7) and the following dimensionless variable (8) the related definitions can be defined (9), (10), and (11):

(11)

(7)

(10)

(9)

(8)

These terms coupled produce a compact version of Schrödinger’s equation:

(12)

This final equation (12) has concluded in a two-dimensionless differential equation to solve. It represents two systems for the eigenvalue problems, but can still be solved with the same single-electron system algorithm with the minor compact addition.

2. Discussion of Methods

2.1 Preservation of Orthogonality

A key aspect of this method is the preservation of the associated orthogonality and this starts with the transformation:

(13)

Where U is an orthogonal matrix, is defined as a unitary transformation

We will show that the unitary transformation of (13) preserves the orthogonality of the vector, which can be tested with the dot product . The dot product after the unitary transformation should stay the same.

This verifies that orthogonality is preserved and the systems holds with the expected results.

2.2 Jacobi Method for Diagonalization

The simplest form to describe the matrix needed for solving is as follows:

(14)

Where *u* is the eigenvector representing the wave function corresponding to its eigenvalue energy level, and is the matrix containing the discretized Schrödinger’s equation.

The most important first step in algorithmic mathematical development is to discretize the second derivative. The defined domain of 0 to needs to be restricted with a maximum value of decided to be 7, which should suffice for this case. The step size is fixed by and this allows the discretized form of Schrödinger’s equation to be written:

Where is the harmonic oscillator potential either alone or with the addition of the Coulomb potential term

By writing out the matrix through the use of (15), the potential terms can be formed into a tridiagonal matrix

(15)

By applying rotations to the matrix in order to make it become diagonal the following equation (16) is sequenced:

(16)

With the matrix *S* being in the form of:

This allows for the triple matrix product to be computed to derive a simple multiplication scheme. By calling the elements of the transformed matrix and let *k* and *l* be the indexes of the largest non-diagonal element the following equations are reduced:

Using these derived iteration-based equations, the next step is after every rotation, to also rotate the eigenvectors. Using (16) through the use of the orthonormal vectors reduce into the following explicit rotation results:

This concludes the analysis of Jacobi Method for Diagonalization and allows the progression towards algorithmic development.

3. Discussion of Algorithms

3.1 Jacobi’s Rotation for Tridiagonal Toeplitz Matrix

The core algorithm for this project involves a few key areas that are highlighted in their respective sections below.

**Solve Matrix**

**def** solve(A, R, tol):

n = A.shape[0]

iterations = 0

maximum, k, l = find\_max(A)

**while** maximum > tol:

iterations += 1

rotate(A, R, k, l)

maximum, k, l = find\_max(A)

**return** iterations, tol

It can be seen that the matrix is rotated and subsequently fed into the find\_max function until the desired tolerance precision is reached. The find\_max and rotate functions can be seen below. The iteration count is also outputted per the request of the prompt for later analysis.

**Find Max Non-Diagonal**

**def** find\_max(A):

n = A.shape[0]

maximum = **abs**(A[0,1])

max\_k=0

max\_l=1

**for** i **in** **xrange**(n):

**for** j **in** **xrange**(i+1, n):

**if** **abs**(A[i,j]) > maximum:

maximum = **abs**(A[i,j])

max\_k = i

max\_l = j

**return** maximum, max\_k, max\_l

**Rotate Matrix**

**def** rotate(A, R, k, l):

n = A.shape[0]

tau = (A[l,l] - A[k,k])/(2\*A[k,l])

**if** tau > 0:

t = 1./(tau + math.sqrt(1 + tau\*\*2))

**else**:

t = 1./(tau - math.sqrt(1 + tau\*\*2))

c = 1 / math.sqrt(1+t\*\*2)

s = c\*t

a\_kk = A[k,k]

a\_ll = A[l,l]

A[k,k] = c\*\*2\*a\_kk - 2\*c\*s\*A[k,l] + s\*\*2\*a\_ll

A[l,l] = s\*\*2\*a\_kk + 2\*c\*s\*A[k,l] + c\*\*2\*a\_ll

A[k,l] = 0

A[l,k] = 0

**for** i **in** **xrange**(n):

**if** i != k **and** i != l:

a\_ik = A[i,k]

a\_il = A[i,l]

A[i,k] = c\*a\_ik - s\*a\_il

A[k,i] = A[i,k]

A[i,l] = c\*a\_il + s\*a\_ik

A[l,i] = A[i,l]

r\_ik = R[i,k]

r\_il = R[i,l]

R[i,k] = c\*r\_ik - s\*r\_il

R[i,l] = c\*r\_il + s\*r\_ik

To solve the Part B case (one particle system) the basic matrix generation is showcased as follows:

**Part B: One Particle System**

**def** make\_matrix\_case\_b(n, omega, p\_max):

A = np.zeros(shape=(n,n), dtype=np.float64)

rho\_0 = 0

rho\_n = p\_max

rho = np.linspace(rho\_0, rho\_n, n+2)[:-1] # quickfix

h = rho[1]-rho[0]

V = np.zeros(n+1)

V[1:] = omega\*\*2\*rho[1:]\*\*2

d = 2/h\*\*2 + V

e = -1/h\*\*2

A[**range**(n), **range**(n)] = d[1:]

A[**range**(1, n), **range**(n-1)] = e

A[**range**(n-1), **range**(1, n)] = e

**return** A, rho

3.2 Unit Testing

To ensure that our algorithm was working correctly in every execution, unit tests were devised. The first test compares the resulting eigenvalues with the built in numpy .eigh function to ensure they are equivalent to 8 decimal points. The second test uses the numpy .eig function and implements a sorting function to match the results and compares to make sure they are equivalent to the same precision as .eigh and the original algorithm that was developed. If these tests do or do not pass, the user it notified in the command line or text file of the pass or failure.

3.3 Modifications for Specific Mathematical Cases

To solve parts d and e of the prompt, the harmonic oscillator potential and repulsive Coulomb interaction needed to be added respectively. This is fairly trivial and the excerpts from the modified make\_matrix functions are shown below:

**Make Matrix for Specific Cases**

# Case D - Added Harmonic Oscillator Potential

V[1:] = omega\*\*2\*rho[1:]\*\*2 + rho[1:]\*\*2

d = 2/h\*\*2 + V

e = -1/h\*\*2

# Case E - Repulsive Coulomb Interaction

V[1:] = omega\*\*2\*rho[1:]\*\*2 + 1/rho[1:]

d = 2/h\*\*2 + V

e = -1/h\*\*2

4. Discussion of Results

4.1 Part B

How many similarity transformations are needed before you reach a result

where all non-diagonal matrix elements are essentially zero?

Try to estimate the number of transformations and extract a behavior as function of the dimension- ality of the matrix. Compare your results with the analytical ones.

Timing benchmarks

Solutions compared with analytical

It can be noted that the LU Decomposition could only be run to matrix size 10000 x 10000 due to the terminal locking upon the computational load of a 100000 x 100000 or larger matrix.

By analyzing the graphs, it can be seen the quick convergence to the exact solution. This validates the stability and junction reliability of our numerical solutions.

4.2 Part C

Explain Units tests the use of (8)

(8)

The results can be seen below in *Table 1*.

|  |  |
| --- | --- |
| *N* | *ε* |
| 10 | -1.179698 |
| 100 | -3.088037 |
| 1000 | -5.080052 |
| 10000 | -7.079268 |
| 100000 | -9.079101 |
| 1000000 | -10.162831 |

**Table 1: Relative Error for Tridiagonal Special Algorithm N Range**

When trying to increase n to n = 107 the program produces the following error: *divide by zero encountered in log10*. This stems from the fact that with floating point round off error we have essentially created a matrix that is trying to divide elements by zero. To solve this, we would need to make zero a really small number, however this can quickly lead down a rabbit hole of problems including round off error growing out of control and is not recommended.

4.3 Part D

Study the results as functions of the number of integration points N and your approximation to ρmax. The analytical results with our scaling for the one-electron energies are λ = 3, 7, 11, 15, . . . . How many integration points do you need in order to reproduce the analytical results with say four leading digits after the decimal point?

*Table 2*.

|  |  |  |  |
| --- | --- | --- | --- |
| N | General (s) | Specialized (s) | LU (s) |
| 10 | 0.000135 | 0.000069 | 0.000172 |
| 100 | 0.000250 | 0.000155 | 0.000384 |
| 1000 | 0.001889 | 0.001130 | 0.025779 |
| 10000 | 0.028790 | 0.012470 | 14.001648 |
| 100000 | 0.196613 | 0.118152 | Not Tested |
| 1000000 | 1.920405 | 1.172803 | Not Tested |

**Table 2: Computation Time Benchmarks**

A few notes about the system running the programs before analyzing the results should be helpful in gauging performance between machines as seen in *Table 3*.

|  |  |
| --- | --- |
| Operating System | Mac OS X |
| Processor | i7 - 2.2 GHz |
| Memory | 8 GB 1600 MHz DDR3 |
| Hard drive | 128 GB SSD |

**Table 3: System Specifications**

With the test conditions being established we can now dive into the data recorded. It can be observed that the general algorithm takes approximately twice as long to compute as the specialized tridiagonal algorithm. The LU factorization and its large amount of computational time come as no surprise due to it running in Θ(n3) time. These results correlate with our expected findings and validate the program is running properly.

4.3 Part E

With no repulsive Coulomb interaction you should get a result which cor-

responds to the relative energy of a non-interacting system. Make sure your results are stable as functions of ρmax and the number of steps.

Comment the results for the lowest state (ground state) as function of varying strengths of ωr.

For specific oscillator frequencies, the above equation has answers in an analytical form, see the article by M. Taut, Phys. Rev. A 48, 3561 (1993).

5. Conclusion

Through this project, the benefits of developing specialized algorithms instead of always using built in functions have been discovered. The case Av = s to solve the single-dimension Poisson’s equations provided a perfect example to experiment with. The tridiagonal matrix is a unique case in that it has the ability to transcend across a multitude of algorithm solution possibilities. When comparing the specialized algorithm with LU decomposition, their results were almost equivalent, but LU decomposition was much slower in executing the result. If we take into account that they are both *reasonably* fast for small n values, it can come down to how much of a repeated task the algorithm needs to complete. This can come down to how much time the programmer has, since it was much quicker to use the LU built in functions that program a custom algorithm. For large n value datasets, there is no doubt that specialized algorithms are the way to go. This project culminates in fresh perspectives and the learning of new skillsets pertaining to Computational Physics.

6. Future Work and Thoughts

A few ideas I had while working on this project came to me during the process. I would like to try and build three lu functions that use no pivoting, partial pivoting, and full pivoting to solve the tridiagonal matrix. I think it would be intriguing to see the results and possibly find ways to manipulate it to optimize the lu pivot types for situations like this.

I also dove a little deeper into specialized tridiagonal solvers as this project peaked my curiosity into what else was out there. Article [1] was very well optimized to run in parallel which would greatly benefit a problem like this due to its inherent symmetry. Article [2] could allow possible multiplexing across parallel cores to solve the linear system by splicing the right-hand side. If I get time in the future I hope to implement these ideas and compare them to the results achieved in this project.

7. References

[1] Bar-On, Ilan, et al. "A Fast Parallel Cholesky Decomposition Algorithm for Tridiagonal

Symmetric Matrices." SIAM Journal on Matrix Analysis & Applications, vol. 18, no. 2,

Apr. 1997, p. 403.

[2] Terekhov, Andrew V. "Parallel Dichotomy Algorithm for Solving Tridiagonal System of

Linear Equations with Multiple Right-Hand Sides." Parallel Computing, vol. 36, no. 8,

Aug. 2010, pp. 423-438