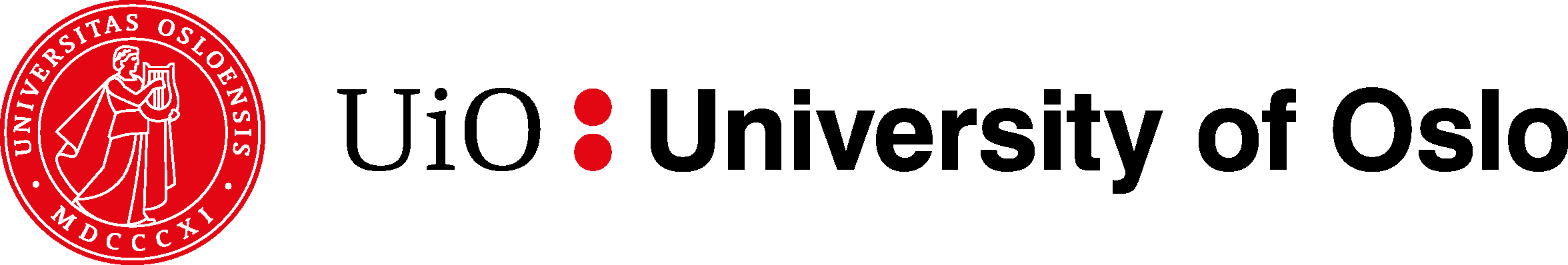
Computational Physics

Project 3



Hunter Wilhelm Phillips (*hunterp*)

October 24, 2018

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

**Abstract**

This project explores the implementation of different integration methods to build a working model of the solar system. Object oriented programming is utilized to develop a class structure for the simulation. The computation of the celestial bodies orbits was the core part of the algorithmic development. This requires the integration of Newton’s law of motion and for this project; Forward Euler and Velocity Verlet were chosen. The project started with modeling a two-body system (Earth-Sun) and progressed until a complete solar system was simulated. Throughout the development, multiple facets of the system were verified including conservation of energy, determining stable velocity for circular orbit, and time step overshoot analysis. Benchmarks were ran comparing the two chosen integrators, which resulted in showing the inaccuracies of the Forward Euler method. Various experiments were conducted with the algorithm such as changing the mass of Jupiter then analyzing the result on the associated celestial bodies and calculating the perihelion precessions of Mercury verifying that there are indeed 43. The results confirmed that an integrator and Newton’s law of motion coupled with object orientation produces a stable simulation of the solar system.

1. Discussion of Problem Statement

In this project, the goal is to successfully simulate the solar system through the use of Newton’s law of motion. To achieve this two integrators were chosen, Forward Euler and Velocity Verlet to solve the coupled differential equations (DE). First, the earth-sun system is set to be analyzed with the resulting discretized DE’s being used as the basis for computation. Second, the algorithms are to be tested and tuned for optimal performance. This includes transferring variables into class structures to transition to object orientation. To produce a circular orbit, the initial value for velocity must be changed to allow for stability, even with varying time steps. To verify that the conservation of energy and angular momentum is held; kinetic, potential, and angular momentum are to be calculated for analysis. A comparison between Forward Euler and Velocity Verlet is to be completed resulting in the decision to only use Velocity Verlet in the rest of the project. Escape velocity is next to be calculated and a custom β modifier is experimented with to change the gravitational force. A three-body problem is simulated next that includes the Sun, Earth, and Jupiter with varying masses of Jupiter to send the system into a disarray. A final model of the solar system that includes all planets is then modeled to verify the algorithm successfully handles the celestial bodies in our solar system. Finally, the perihelion precession of Mercury is to be calculated through the use of a relativistic correction of Newton’s gravitational force and comparing it to the uncorrected original equation. Through this, the challenges at hand have been laid out and the theory and methods explaining the solution are explained in the following section.

2. Discussion of Theory & Methods

2.1 Newtonian Gravitation

When studying the solar system, the core of movement is held within gravity and the associated equations. The classical law of gravitation is given by Newton’s law as seen in (1):

(1)

Where is the gravitational force between two bodies

is the gravitational constant

and are the masses of the two bodies

is the distance between the two bodies

This equation can be broken into components when viewed in three dimensions. By utilizing Newton’s second law of motion, the following individual equations can be derived for the acceleration due to gravity on a chosen body [1].

(2)

These set of equations seen in (2) can be individually integrated to determine the position of the celestial bodies at any provided time. It can also be seen that converting (1) into a vectorial equation in multiple dimensions, shown in (3), allows for discretized calculations in two dimensions as chosen for this project.

(3)

where is the position of the body in the cartesian coordinate system

2.2 General Relativity

The theory of general relativity introduced by Albert Einstein is currently the description of gravity accepted by modern physics. One such anomaly that cannot be explained is the perihelion precession of Mercury’s orbit being 43 arc-seconds per century. This unique case was explained through general relativity by Einstein.

One important aspect of Newton’s law of gravitation is the factor used in closed elliptical orbits. When changing this factor, the associated orbit is altered as well. Instead of computing a space-time manifold to make the proper correction, a general relativistic error term is added to (1), resulting in (4).

(4)

where is the mass of Sun

where is the mass of Mercury

is the distance between Mercury and the Sun

is the orbital angular momentum per unit mass

c is the speed of light in vacuum

By using (4) and coupling it with (5) allows for easier location of Mercury’s perihelion by using the arctangent of the cartesian coordinate frame to calculate the angle of perihelion.

(5)

2.3 Units

Being that this problem is heavily based on an abstract set of units it was decided to expand upon the classifications used in this project. All celestial body masses are expressed as a fraction of solar masses as seen in an example below:

All units related to time in the algorithm other than benchmarking will be in Earth years (). All units of distance will be in astronomical units (). This in turn identifies the velocity to be for this project.

By changing the variables units, the gravitational equations must be modified to accommodate altered velocities and gravitational constant . A simple example case can be seen below using the sun and earth as the celestial bodies.

(6)

Modification of (6) assuming Earth’s orbit is circular:

(7)

Combining (6) and (7) produces:

(8)

Knowing that with unit statement above and , (8) can be reduced into the set of derived units showcased in (9):

(9)

2.3 Discretizing Differential Equations

To derive the foundation for the associated numerical methods used in the algorithm, the following Taylor expansions are formed.

(11)

(10)

2.3.1 Forward Euler’s

Using the leading terms of (10) and (11) coupled with making , (12) and (13) can be obtained:

(13)

(12)

The derived equations (12) and (13) can be discretized into (14) and (15):

(14)

(15)

2.3.1 Velocity Verlet

Using forward difference, can be approximated as:

(16)

By placing (16) into (11) and coupling with , (17) and (18) are produced:

(18)

(17)

Due to the nature of the problem only dealing with gravitational forces, (15) can be used to find resulting in the ability to use (16) for solving.

The derived equations (17) and (18) can be discretized into (19) and (20):

(14)

(20)

(19)

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 System Specifications

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 1*.

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

**Table 1: System Specifications**

With the test conditions being established we can now dive into the data recorded.

4.2 Part B

The returned eigenvalues come out as expected within rounding error for the test case of n = 4, omega = 1, and rho max = 2 with a tolerance of (1E-10) and can be seen in in *Table 2*.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | E1 | E2 | E3 | E4 |
| Numerical | 3.4363526473 | 9.9109886642 | 17.624738783 | 23.827919904 |
| Expected | 3.43635 | 9.91099 | 17.62470 | 23.82800 |

**Table 2: Numerical vs Expected – Part B**

The relationship between matrix dimension and then computational time with the amount of transformational iterations needed for the solution tolerance can be seen below in *Table 3*.

|  |  |  |  |
| --- | --- | --- | --- |
| n | Computation Time (s) | Iterations | Iterations / Element |
| 4 | 0.000360 | 20 | 1.25 |
| 16 | 0.037828 | 440 | 1.72 |
| 64 | 5.868330 | 7408 | 1.81 |
| 128 | 90.74790 | 30064 | 1.83 |

**Table 3: Comparing Relationships for Growing n x n Matrix Size in Part B**

The tolerance that was selected (1E-10) to be zero for this algorithm and this resulted in the values shown in *Table 3* being produced. It is interesting to note that the larger n grows, the less iterations per element are needed, it is just a case of so many iterations being needed that grows the computational time so large.

4.3 Part C

The implementation of the unit tests described in Section 3.2 result in the following output either on the command line or file, depending on the script

Part B Sol numpy .eig numpy .eigh

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

3.4363526473 3.4363526473 3.4363526473

9.9109886642 9.9109886642 9.9109886642

17.624738783 17.624738783 17.624738783

23.827919904 23.827919904 23.827919904

The Unit Tests have PASSED

It allows the end user to individually see the comparison and get an explicit Passed or Not Passed at the end of the output to validate their visual observation.

4.4 Part D

Through the exploration of quantum dots in three dimensions for one electron the slightly modified algorithm resulted in the following expected vs numerical results in *Table 4*. The same initial conditions used in Section 4.2 were used for these values.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | E1 | E2 | E3 | E4 |
| Numerical | 4.3444801411 | 11.172505935 | 18.858022617 | 25.224991306 |
| Expected | 4.35 | 11.18 | 18.86 | 25.25 |

**Table 4: Numerical vs Expected – Part D**

The relationship between matrix dimension and then computational time with the amount of transformational iterations needed for the solution tolerance can be seen below in *Table 5*.

|  |  |  |  |
| --- | --- | --- | --- |
| n | Computation Time (s) | Iterations | Iterations / Element |
| 4 | 0.000288 | 16 | 1 |
| 64 | 5.524490 | 7407 | 1.81 |
| 128 | 77.13050 | 29884 | 1.82 |

**Table 5: Comparing Relationships for Growing n x n Matrix Size in Part D**

The algorithm is slightly faster for this case when compared to Part B, but that is just an observation for an unrelated facet.

4.5 Part E

Through the continued divergence of quantum dots in three dimensions for two electrons the slightly modified algorithm resulted in the following varying value results in *Table 6*. To quantify this difference the results in [1] are used to compare. The same initial conditions used in Section 4.2 were used for these values.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | E1 | E2 | E3 | E4 |
| = 0.01 | 3.5300584925 | 10.040939887 | 17.789212852 | 23.848602101 |
| = 0.50 | 3.8334295107 | 10.360535871 | 18.108435348 | 24.105932602 |
| = 1.00 | 4.6851803654 | 11.320699653 | 19.057912350 | 24.944540964 |
| = 5.00 | 15.911354332 | 30.837117967 | 49.957335918 | 78.502525115 |

**Table 6: Varying**

Through the comparison it can be seen the increasing affects the Energy directly and can be correlated with verification.

5. Conclusion

Through this project, the differing solution spaces for Schrödinger’s equation have been understood. It also verifies that some quantum mechanical systems can be solved through the use of diagonalization algorithms. The highly stable Jacobi algorithm displayed its reliable use, but with the cost of computation time. Typically, solution spaces less than n = 64 do not occur and above that the algorithm would take forever to solve large data sets. The behavior discovered in the modified special cases is expected and provides a broader scope with the use of eigenvector algorithm solvers. The project concludes in a more complete understanding of eigenvalue problems and the associated algorithm development to solve them.

6. References

[1] Hjorth-Jensen, Morten. "Project 3." *Computational Physics FYS3150*, Fall 2018. University of Oslo. Lectures.