Workshop in Computational Mathematics
N-Body-Simulation Project

Alon Oved
Tel-Aviv University

June 22, 2020

1 Introduction

The project objective was to implement a solution to the N-Body-Simulation problem. In physics, an N-Body problem is a simulation of a dynamic system of particles under the influence of physical forces such as gravity. Using this broad definition, the types of particles that can be simulated using N-Body methods are quite significant, ranging from celetial bodies to individual atoms in a gas cloud. In this work our particles would be planets (and a star - the Sun) in the Solar System under the influence of gravity, limited to 2 or 3 dimension. This problem can be generalized to N bodies in M dimensions with any collection F of forces interacting between them. For the 2-body problem, there exists a closed formulated solution which we implemented and used as a reference for accuracy analysis. All project files including python records, data files, Solar System animation videos (2D and 3D) are included to the repository
https://github.com/Mathematics-project/n-body-simulation

2 Formulation

N-Body-Simulation problem is essentially a dynamic continuous problem over finite time. Computing plantes' location over time under gravitational interactions is limited both to discrete steps and simulation time. For a constant simulation period, a small time step benefits precision but requiers more steps ,whereas a bigger time step will cost in accuracy and demands less steps to simulate. The time step, noted h in below formulation and in code files, was adjusted by experiments that will be discussed later. The N-Body problem considers n point masses m_i , $\quad i = 1, 2, ..., n$ moving under the influence of mutual gravitational attraction. Each mass m_i has a position vector x_i and a velocity vector v_i . Newton's second law states that mass times acceleration, meaning $m_i \frac{\partial^2 x_i}{\partial^2 t}$, is equal to the sum of the forces on that mass. Newton's law of gravity says that the gravitational force felt on mass m_i by a single mass m_j is given by

$$F_{ij} = \frac{Gm_i m_j}{||x_j - x_i||^2} \cdot \frac{(x_j - x_i)}{||x_j - x_i||} = \frac{Gm_i m_j (x_j - x_i)}{||x_j - x_i||^3}$$

Where $G = 6.67 * 10^{-11}$ in $m^3/(kg \cdot sec)$ units is the universal gravitational constant and $||x_j - x_i||$ is the magnitude of the distance between x_i and x_j (metric induced by the l_2 norm).
Summing over all masses yields the n-body equations of motion:

$$a_i m_i = F_i = \sum_{j=1, j\neq i}^{n} \frac{Gm_i m_j (x_j - x_i)}{||x_j - x_i||^3}.$$

Dividing the equation by m_i enables us to retrive acceleration a_i for every $i = 1, 2, ..., n$.
When there are 3 or more planets, there is no general closed solution that can determine the positions and velocities at any given point in time. Instead, the system must be solved numerically given the starting values. For each planet, given that we know the position, we can define its velocity as

$$v_i(t) = \frac{\partial x_i}{\partial t}(t) \quad \text{and its acceleration as} \quad a_i(t) = \frac{\partial v_i}{\partial t}(t) \ .$$

3 Algorithms

In this work we use Euler and Leapfrog algorithms for solving the problem and analyse their results over time and over different step size h. Both algorithms consider all pairs and so their time complexity is $\Theta(n^2)$ where n is the number of planets.

3.1 Euler Algorithm:

The Euler method is a 1st order Taylor approximation. For any mass m_i update location and velocity:

$$x_i(t+h) = x_i(t) + h \cdot \frac{\partial x_i}{\partial t}(t) = x_i(t) + h \cdot v_i(t)$$

$$v_i(t+h) = v_i + h \cdot \frac{\partial v_i}{\partial t}(t) = v_i(t) + h \cdot a_i(t)$$

Integrating over finite time T, the overall error is proportional to h, meaning it is a first order method.

3.2 Leapfrog Algorithm:

The leapfrog method updates the velocity in the midpoint of the interval instead of at its end (see Figure 1), i.e.

$$x_i(t+h) = x_i(t) + h \cdot \frac{\partial x_i}{\partial t}(t+h/2) = x_i(t) + h \cdot v_i(t+h/2)$$

$$v_i(t+3h/2) = v_i(t+h/2) + h \cdot \frac{\partial v_i}{\partial t}(t+h) = v_i(t+h/2) + h \cdot a_i(t+h)$$

The overall error is proportional to h^2, Leapfrog is therefore a second order method. Notice that the method requires a velocity update before the first step occurs, because the first location update is:

$$x_i(0+h) = x_i(h) = x_i(0) + h \cdot v_i(h/2)$$

That update requires $v_i(h/2)$ while we only have a starting velocity value at $v_i(0)$. The simplest approximation is just to do a single half step of Euler and compute $v_i(h/2) = v_i(0) + 0.5 \cdot h \cdot a_i(0)$ Although this is not a midpoint method, and so has an error of h^2, we only do this once so it does not lower the order of the method, which remains second order. Some advantages for using Leapfrog algorithm:

- Time reversal invariant: Newton's equations of motion are invariant under time reversal and so is the method.
- Conserves Angular Momentum: In symmetric potential, angular momentum is conserved and leapfrog algorithm conserves it exactly.

Unfortunately, the other quantity conserved by Newton's equations, Energy, is not exactly conserved by the algorithm.

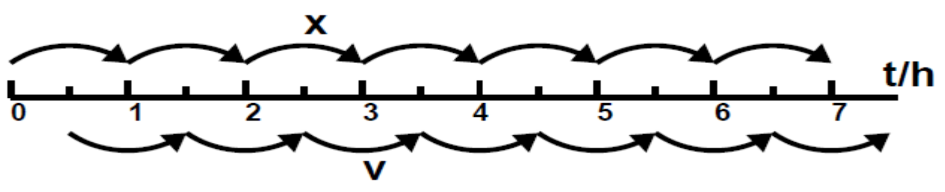


Figure 1: Leapfrog

4 Two-Body Simulation - Exact Solution

– From Yoel's reference notes –
Let m_1 and m_2 be two masses with initial positions $x_1(0)$ and $x_2(0)$, and initial velocities $v_1(0)$ and $v_2(0)$ respectively. We want to find the positions of the two masses at time t, that is $x_1(t)$ and $x_2(t)$. Since our system has only two masses and no external forces, by Newton's third law, $F_1 = -F_2$ and thus $a_{cm} = 0$. This means that the center of mass has no acceleration, namely, it moves at a constant speed. We then change our coordinate system to be centered at the center of mass, denote by l_1 and l_2 the new coordinates. Since x_{cm} is the center of mass, in our coordinates $m_1 l_1 + m_2 l_2 = 0$. Thus, knowing how l_1 evolves with time immediately tells how l_2 evolves with time. We rename l_1 as R and denote $r = R/||R||$. By Newton's second law the acceleration of m_1 is given by $a = -\frac{Kr}{||r||^2}$. Where $K = \frac{Gm_2^3}{(m_1+m_2)^2}$. Instead of solving for $r(t)$ we will solve r in polar coordinates, that is $r(\theta)$. Following the details in the notes, the equation that needs to be solved is $t(\theta) = \int_0^\theta \frac{r^2(\theta)}{L} d\theta$. Where L is determined by the initial conditions. For every time t, we solve numerically that equation for θ. m_1 and m_2 new locations are

$$x_{1,new} = (\, r(\theta)\cos(\theta)\,,\, r(\theta)\sin(\theta)\,) + (x_{cm} + v_{cm} \cdot t)$$

$$x_{2,new} = \frac{-m_1}{m_2}(\, r(\theta)\cos(\theta)\,,\, r(\theta)\sin(\theta)\,) + (x_{cm} + v_{cm} \cdot t)$$

5 Collecting Data

Our implementation requires 2D imformation of each participant planet, meaning a starting location vector $(location_x, location_y)$ and a velocity vector $(velocity_x, velocity_y)$. The starting location vector was initialized with $(r_i, 0)$ where r_i is planet i mean distance from the Sun in meter units. The velocity vector was initialized with $(0, v_i)$ where v_i is planet i mean orbital speed around the Sun in meter/second units. In addition, we added the planet's mass in kg units. Lastly, for 3D simulation, we included an inclination value which is the tilt of the planet's orbit around the Sun and expressed as the angle between the Sun-Earth reference plane and the planet's orbital plane. All those items were taken from Wikipedia page
https://en.wikipedia.org/wiki/List_of_gravitationally_rounded_objects_of_the_Solar_System

6 Dimension Compatibility

Both algorithms employed in our work are suitable for any dimension $d > 1$ due to their vectorized formulation. However, the python program performs a simulation as well as animation only in two or three dimensions. That said, with minor changes the program can be altered to simulate in any dimension.
In order to transform our 2D data to 3D data we used Rodrigues' rotation formula. If v is a vector in $\mathbb{R}^3$ and k is a unit vector describing an axis of rotation about which v rotates by an angle θ according to the right hand rule, the Rodrigues formula for the rotated vector is $v_{rot} = Rv$

$$\text{Where } R = I + (\sin\theta)K + (1-\cos\theta)K^2 \quad \text{and} \quad K = \begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix}$$

In our case, the vector $k = (0, 1, 0)$ and to plug that formula in our 2D data we added to the location and velocity vectors a third coordinate and initialized it to zero. That is, only if required to perform a 3D simulation.

7 Code breakdown

The project was implemented in python3.7 and includes the files:

1. Math_Project.py: the main file, includes objects for the planet and for the n-body-simulation. Contains also all functions required to perform a simulation with a specific data file, a chosen algorithm, time step and simulation time.

2. Performance_Analysis.py: consists functions for analyzing simulations with different inputs (data files, dimension requirements, algorithm, time step etc), fuctions to compare algorithms accuracy and efficiency and functions to plot the results in appropriate graph.

3. Solar_System_2D_Animation.py: Produces a 2D video for a data file input with a desired simulation time, algorithm and time step.

4. Solar_System_3D_Animation.py: Produces a 3D video for a data file input with a desired simulation time, algorithm and time step.

5. Cython files - Math_Project_cython.pyx, setup.py : Instructions for using Cython are in setup.py file.

Required packages: Numpy, Mpmath, Matplotlib

8 Results

8.1 Number of Simulation Steps per Second

The initial construction of the simulation was based on 5 - 6 functions called from one another. The result was $\sim 18,000$ simulation steps per second (sps) in a two-body simulation and $\sim 2,100$ sps for six-body simulation. As expected, the 3 times increase in number of planets resulted in ~ 9 times the time of simulation for the same data file. After code profiling, it became clear that the numerous functions design lacks efficiency and we reduced the number of active functions. Additionally, the built-in numpy norm function was marked by the profiler as a hot spot in a specific line, so we changed the norm implementation to a manual calculation based on coordinates. Those changes increased the program performance by a signuficant margin to 26,000 sps for two-body and 3,600 sps for six-body, 44% and 71% increase respectively. The last and fastest version of our code was impoved by Cython tool. "Cython is a programming language that makes writing C extensions for python as easy as python itself" (from their website). The source code was translated into optimized C code and compiled as Python extension module. That resulted in 31,000 sps for two-body and 4,000 sps for six-body, another 18% and 11% performance upgrade respectively. Results presented are for 2D simulation, 3D simulation takes $\sim 8\%$ extra time to perform the same amount of simulation steps both in 2 and 6 body simulation.
Notice that the difference between two-body and six-body simulation is 7.75 times, less than the expected 9 factor. Possible causes for that could be somewhat small amount of simulation steps or some computational optimization under-the-hood of numpy package when using arrays and matrices.
During the search for efficiency improvements, some efforts have not added value and are not a part of the final work, let us review some of those tools.

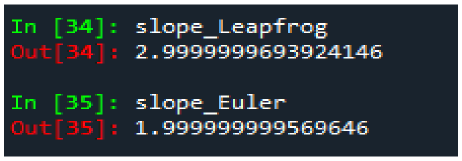
1. **pypy**
   pypy is a Python inerpreter designed to enhace performance. We discovered that regarding our work, pypy has not made our program run faster. The contrary, the program's performance was ~ 3 times slower with pypy as oppose to the built-in python inerpreter.

2. **Numba**
   Numba is a high performance python compiler. currently, numba does not work with the numpy package which is heavily used in our code, as a result we skipped efforts to fit our files to Numba.

3. **Nested for loops order**
   The most visited function for any simulation is a_lst, which computes the total interacion forces over all planets in the simulation. There are two nested for-loops in that function. We changed the order of them to better understand the more efficient way to make the necessary calculations. We created a 1000 planets file with random data to check that, and discovered that the original order was running $\sim 10\%$ faster than the changed one. Obviously, we decided to keep the original order.

8.2 Asymptotic behavior - Euler vs Leapfrog

Verifying that our implementation is correct was a challenge, so we wanted to confirm first that our implementation supports the theory we know for Euler and Leapfrog algorithms. As stated, Euler is a first order method while Leapfrog is a second order method. That means that for Euler method the error for a single step is $\Theta(h^2)$ and simulation time t ,meaning t/h steps, the error is $\Theta(h)$ - first order method. For Leapfrog algorithm, error for one step is $\Theta(h^3)$ and for time t is $\Theta(h^2)$. Error was taken with respect to the reference and after taking log on both Error and h axis, the slope for Euler was 1.999, for Leapfrog was 2.999 as expected (see Figure 2).


```
In [34]: slope_Leapfrog
Out[34]: 2.9999999693924146

In [35]: slope_Euler
Out[35]: 1.999999999569646
```

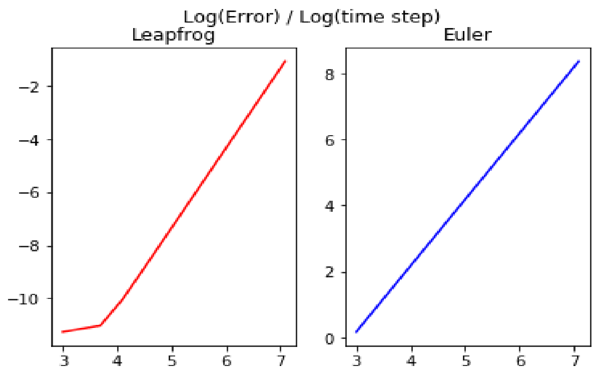


Figure 2: Asymptotic behavior

8.3 Error over time - Euler vs Leapfrog

Simulated with Sun-Earth data file, over two years of data with time step $h = 1_{sec}$ the results stressed a profound differences regarding algorithms accuracy (see Figure 3). While Euler method diverged completely from Earth's reference location with $\sim 7,000_{km}$ error, Leapfrog method stayed as close as half a meter error from Earth's reference location.

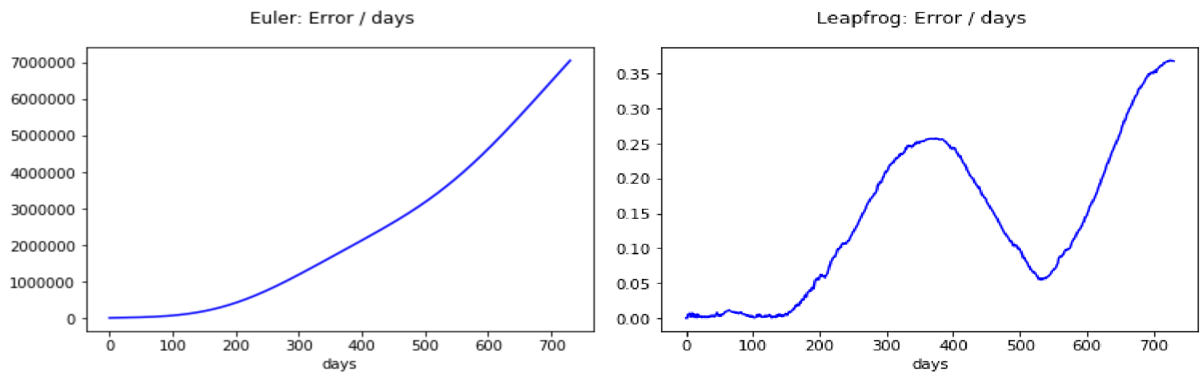


Figure 3: Euler vs Leapfrog

8.4 Time step refinement

The best version of our code requires $\sim$ 17 minutes to simulate Sun and Earth orbit for a year with time step of $h = 1_{sec}$, and $\sim$ 130 minutes for 6-planet simulation. Leapfrog's results with that time step were impressive and we wanted to find out how error grows with a larger time step. Taken h between 100 - 400 over 4 year simulation showed that the relative error did not surpass 10^{-8} factor, still impressive. When time step multyplied by 10 to the 3000 seconds, the relative error rose to 10^{-5} factor which we decided is too high for us. The chosen time step for all later simulations was $h = 300_{sec}$. One year simulationwith that h for two planets takes $\sim$ 5 seconds and $\sim$ 30 seconds for six planets.

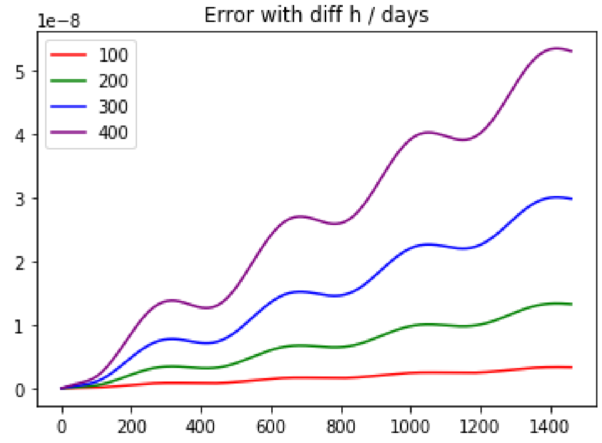


Figure 4: Relative Error over 4 years of data

8.5 Solar System Animations

We created two python animation files, one for 2D and one for 3D. Both files require an appropriate data file, simulation time and a name for the mp4 file saved to folder. 2D animation was created with 5 planets (Sun, Earth, Mercury, Venus, Mars) and 6 planets (Jupiter included) for several Earth-years. Orbit inclination seen in 3D only is almost unnoticeable because angles are pretty small $(0^{\circ} - 7^{\circ})$. 3D animation generated only with the 5 planet file, We did not animate on 3D the six planet file because Jupiter, a gas giant, is too far from the rest of the four terrestrial planets causing a space stretch. That stretch causing a quality degradation to the video.
All mp4 files were added to the Github repository and many other animation videos can be created with other data files.

9 Possible Extensions

We recognize that there are some improvements that can be made in performace, research and implementation aspects.

1. **Additional Algorithms**
   We implemented the Leapfrog and Euler algorithms, but there are other methods for N-Body problem, such methods include FR and PEFRL algorithms which are fourth order methods, meaning their error is $\Theta(h^4)$ for time step h. On one hand, both algorithms are more accurate, as Euler and Leapfrog are first and second order methods respectively. On the other hand they are much slower with about 4 times more calculations required at each step. If accuracy is a priority, it would be wise to consider using one of them.
   Barnes-Hut simulation method is an approximation algorithm, it divides space into cubic cells and only interactions between particles from nearby cells need to be treated individually. Particles in distant cells can be treated collectively as a single large particle centered at the

distant cell's center of mass. This methos yields optimal $O(n \log n)$ time complexity whereas both algorithms we used had $\Theta(n^2)$ time complexity.

2. **Implementation in C language**
   As the use of Cython showed, the code can run faster if some (maybe all) of the code would be written in C. We suggest implementing the functions calculating the simulation steps in C and merge it with the analysis code written in python. This should result with optimized files and faster simulations.

3. **Two-Body vs N-Body Analysis Gap**
   The Reference location we had was a closed solution for the 2-body problem. All conclusions and analysis were based on the Sun-Earth data which creates a gap of knowledge when discussing the six planet (or any $N \geq 3$) simulation. For example, we know that $h = 300_{sec}$ keeps our error in the 10^{-8} range when using the two-body data, but we can't be sure of that while using the six-planet data file. Although animation shows stability in the system, we can't be certain that it actually is. Moreover, when rising to 3D data that accuracy might be more challenged as higher dimension requires more caculations and possibly more error accumulated. Our suggestion is to set $h = 1_{sec}$ as a reference data for any N-Body data file $(N \geq 3)$. The obvious drawback is the time it takes to simulate a large collection of planets for an extended period and such small time step.

Acknowledgments
This project was guided closely by **Prof. Yoel Shkolnisky** and **Prof. Haim Avron** of Tel-Aviv University.