Draft version December 6, 2020
Typeset using LaTeX default style in AASTeX62

An Exploration of N-Body Simulations

Michael Porritt - z5206336[1]

[1] *https://github.com/porrittmichael/*

1. INTRODUCTION

In the field of computational astrophysics, N-body simulations are one of the most persistent and broadly applied algorithms. The applications range from the simulation of a few planets around a star, up to hundreds of millions of stars in a galaxy, or even simulations of the entire universe and it's evolution.

In this project, I outline the process of N-body simulation and a few different methods that can be applied, and subsequently implement some of the methods in python.

The first well-known introduction of the N-body problem to the realms of science is in Isaac Newton's Famous *Philosophiæ Naturalis Principia Mathematica* in 1687. From here the N-body problem has evolved into the complex computational algorithms that are used today.

1.1. *The Two-Body Problem*

To understand the N-body problem, we must first delve into the more confined two- and three-body problems:

Between 1609 and 1619, Johannes Kepler introduced his laws of planetary motion, based on his observations of the planets and moons of our solar system. A principle result of his observations was the discovery that planets orbited the sun in elliptical paths. In subsequent years, Isaac Newton was able to build upon this idea to develop the laws of motion and the law of universal gravitation in *Principia*. The universal law of gravitation describes the force due to gravity on an object with mass m_1 due to a point mass m_2 as follows:

$$F_G = G\frac{m_1 m_2}{r^2} \tag{1}$$

Where G is the gravitational constant ($\approx 6.67 \times 10^{-11} m^3 kg^{-1} s^{-2}$), and r is the distance between the two objects.

The two-body problem is the subsequent problem to find the paths that two objects will trace out in space given their initial positions and velocities. It turns out that the two-body problem can be solved analytically, and that we can explicitly describe the exact paths of any two objects. The result is that two bodies in space will each trace an ellipse around the common centre of mass. See figure 1.

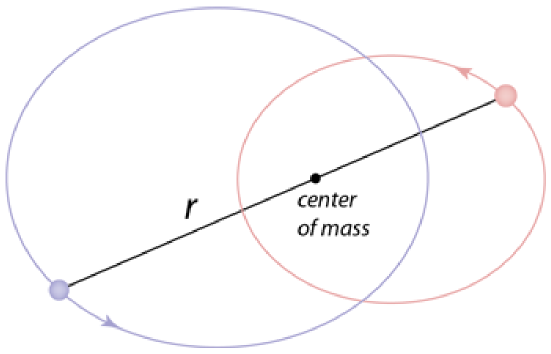

Figure 1. An example of a two-body orbit. Both objects trace ellipses with the centre of mass at one focus. Image from http://hyperphysics.phy-astr.gsu.edu/hbase/Mechanics/keplerd.html.

This is very useful since this can be used to great effect when we observe systems such as the moons of Jupiter, or indeed the planets around the sun. We find that observations of these systems very closely match the two-body solutions to these systems, even though they are not truly two-body systems. This is because, in the case of the solar system, the sun enormously dominates the total gravitational force on any given body, and so the resultant motion is very nearly elliptical, and the analytical two-body solution is a good approximation.

1.2. *The Three-Body Problem*

The three-body problem is where things get interesting. It seems like only a slight increase in complexity from two-bodies, but it turns out that there is no general, formulaic solution to the three-body problem; Determining the motion of three bodies due to gravity is impossible by analytical means (bri 2016).

There do exist some interesting special-case three-body systems that have explicit solutions. Figure 2 shows the classic figure-8 solution and GIFs of some simple three-body choreographies are below (Montgomery 2010):

- Figure 8 solution http://www.scholarpedia.org/article/File:3body_problem_figure_9_eight.gif
- Lagrange equilateral solution http://www.scholarpedia.org/article/File:Lagrange-60.gif

There have been found many unique analytical solutions to the three-body problem, but in practice in astronomy, these are almost always useless, since they rely on very specific masses and starting conditions to generate, and any deviation from such conditions leads to the system becoming chaotic.

The three-body problem has no general solution, therefore the only way to obtain the evolution of a general system of three or more bodies is to use computational methods.

Figure 2. A representation of the simple figure 8 solution to the three-body problem. All three objects are of equal mass and trace out the same path. Images obtained from https://news.ucsc.edu/2019/08/three-body-problem.html.

2. COMPUTATIONAL METHODS

The N-body problem is finding the state at some point in time of a system of N particle-like bodies from the initial state of the system. For the purposes of this project, this is a computational problem, which means that any solutions that we obtain can never be completely true, simply since there does not exist a ”true” solution.

At a high level, the method to computationally solve the N-body problem is very similar to any other physical simulation: We will take the initial conditions, and for a series of time-steps we evaluate the state of the system given the results of previous time-steps. More specifically, at each time-step, we calculate the total gravitational force on each object given their current positions (Eqn. 1). Then, we calculate the acceleration due to this force on each object, and twice integrate the acceleration to find the new position of each object. This process is expressed in pseudo-code below.

The key point of note in this procedure is that the methods to obtain the accelerations and to integrate are left ambiguous. There are several well-known computational techniques for both calculating accelerations and for numerical integration. I will go through a few of these methods in the subsequent sections.

2.1. *Force Calculations*

The first step in each time-step is to calculate the net force due to gravity on each particle due to each other particle. To do this we use equation 1. There are several common ways to do this: the most simple and most accurate being direct summation. However, since this costs $\Theta(N^2)$ calculations each iteration, it is often worth considering cheaper approximations.

Algorithm 1: N-Body Simulation Procedure

```
Result: 2D Array containing the position of each body at each time-step.
$M \leftarrow$ Array of masses;
$V \leftarrow$ Array of initial velocities;
$dt \leftarrow$ Time-step;
$t_{max} \leftarrow$ Simulation time;
$P[0] \leftarrow$ Array if initial positions;
$t = 0$;
while $t < t_{max}$ do
    $A \leftarrow$ GetAccelerations$(M, P[t])$ ;
    $P[t + dt] \leftarrow$ Integrate$(P, V, A, dt)$ ;
    $t = t + dt$;
end
return $P$
```

2.1.1. *Direct Summation*

The most accurate way to calculate the forces on an object is through direct summation. This is the simplest method and requires iteration through all pairs of forces and calculating the gravitational force. This is essentially the brute force method, but can be somewhat improved: From Newton's third law we know that

$$\vec{F_G}(i,j) = -\vec{F_G}(j,i).$$

Therefore, we only need to consider every unique combination of particles. This brings the number of force calculations to

$$\frac{N(N-1)}{2} \approx \frac{N^2}{2}.$$

Where N is the number of bodies.

Therefore, direct summation is computationally expensive in theoretical terms, but the overhead will be minimal. This means that direct summation is best suited for systems with a small N. This might seem to lack value, but in fact this is entirely reasonable if the system in consideration is something like a stellar system, which will usually have N on the order of a single digit. Such a system will usually also require greater a accuracy in any case. Thus direct summation is a perfectly applicable to certain types of simulations, but is often far too expensive for very large numbers of particles due to the $\Theta(N^2)$ time complexity.

2.1.2. *Barnes-Hut Summation*

The Barnes-Hut method (Barnes & Hut 1986) is a very efficient, tree-based method of force summation. In Barnes-Hut we recursively divide the space into 3D cubes. That is, if a cube contains more than one particle, we split it into 8 octants and continue as such until each "leaf" node contains exactly one particle (Koby ????; NW- 2019). Each octant in the tree is then considered to have the total mass and centre of mass of all the particles it contains. An excellent visualisation of this process is shown in figure 3.

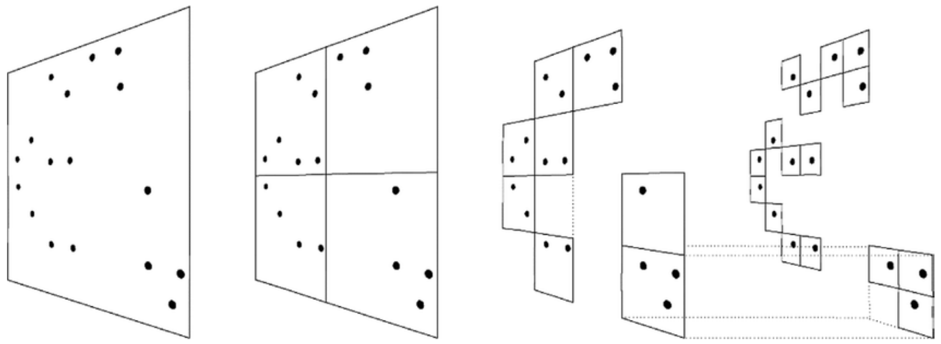

Figure 3. A visualisation of how a Barnes-Hut tree is recursively divided into sub-octants. Springel et al. (2001)

To find the net acceleration at some point, we group the contributions of distant objects into the largest cube that fits within some "opening angle", θ (As in figure 4). This is a clever simplification that allows us to explicitly guarantee the level of accuracy desired for a given simulation. Essentially, we recursively iterate through smaller and smaller cubes, and if a cube fits within the opening angle, we group its contents into a single particle specified by the centre of mass and total mass of the cube, calculated when we constructed the tree.

The primary advantage of the tree method however, is that for a given position, we must compute the gravitational contributions from on average only $log(N)$ 'bodies' - assuming a reasonably large and well-distributed set of bodies. This means the time complexity for each iteration becomes $\Theta(Nlog(N))$. This is an enormous improvement from the brute force method for large N and allows us to choose our desired level of accuracy given by the opening angle.

The main downside to this method is the overhead involved for each iteration; primarily the tree construction which is proportional to $Nlog(n)$ complexity. However, in context of the main applications for this method, namely very large N, any amount of overhead will likely be a significant improvement upon direct summation. In particular, an application worth mentioning is to simulations containing dense systems separated by large empty spaces; for example galaxy mergers (NW- 2019).

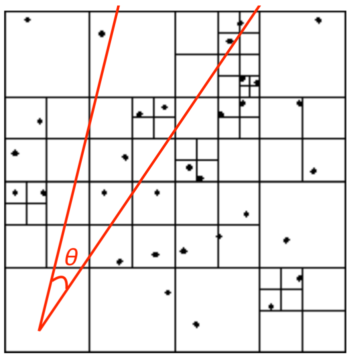

Figure 4. NW- (2019)

2.1.3. *Other methods*

For the extent of this project, I will stop there in my descriptions since my implementation will only include direct summation and Barnes-Hut summation. There are many other methods suitable to a wide range of applications, and even then summation methods can be combined and modified for the sake of speed and accuracy. An excellent description of many more methods than I've been able to outline here is in Graps (1996).

2.2. *Integration Methods*

The second part of each iteration of an N-body simulation is integrating the obtained accelerations to find the state of the system at the next iteration. This involves first integrating accelerations to find the new velocities, and then integrating these to obtain the new positions. Numerical integration is a very well-documented field, so I will choose not to go into much depth or breadth. Following is a description of the two integration methods I implemented along with their advantages and disadvantages.

2.2.1. *Euler Integration*

Euler's method for integration is the simplest form of numerical integration (Barker 2017). It is a first-order integration technique so it is fairly inaccurate and often diverges quite quickly. In context of N-body it's only real value is in it's ease of implementation.

2.2.2. *Leapfrog Integration*

Leapfrog integration is a second-order approach, and it has the unique property of being time-reversible. That is, a state can be considered in reverse time and leapfrog integration will arrive at precisely the previous state. This is a very desirable property for N-body simulation because it is a physical property associated with the conservation of energy and angular momentum (McMillan ????).

3. IMPLEMENTATION

In my implementation of this algorithm, I wanted to make the adding and using of different summation and integration methods as straightforward as possible with one common N-body simulation algorithm. The end result of my implementation can be found on my GitHub at https://github.com/porrittmichael/COMP4121-N-Body-Simulation.

My implementation contains the overhead file `NBodySimulator.py` which outlines the skeleton algorithm 1. An `NBodySimulator` object is assigned a summation class and integration class upon initialisation which must contain the algorithms associated summation and integration respectively. The output of the skeleton program is a matrix of positions. `NBodySimulator` also contains a static `animate` method which animates the particles using `vpython`.

Overall, I consider the program a success: it can simulate and even animate the progression of an N-body system. We do not expect the performance to be particularly good given that python is not a high-performing language, and indeed we find that at only $N = 20$ the computation time is around 22 seconds for even the simplest algorithm (direct summation and Euler integration).

3.1. *Assessing Correctness*

Assessing the correctness of an N-body algorithm is not a simple task, since for the general case there *is* no true solution. There are several methods that would be appropriate to use in order to assess the accuracy of an algorithm:

Two-Body Simulations—We can of course assess accuracy by simulating only two bodies. This way, unlike the general case, there is an easily calculable solution to which we can compare the results of an algorithm.

Special Small-N Simulations—It was mentioned previously that for $N \approx 3, 4, 5$ there are some special-case solutions to the N-body problem. Similar to two-body simulations this would allow us to compare the results of the algorithm to the known solutions.

Decreasing Time-Step—For larger N, a way to assess the validity of a certain algorithm might be to continuously decrease the time-step of the simulation until it converges; our algorithms are approximations, but the more we decrease the time-step of the simulation, the more accurate it should become. For summation methods other than direct summation, other parameters would have to be progressively increased to higher levels of accuracy to ensure convergence (e.g. decreasing the opening angle in Barnes-Hut summation).

Energy and Momentum—This is a very common means of assessing the validity of an N-body simulation. It is less of a computational approach and more of a physical one: Two fundamental laws that a good N-body simulation ought to obey is conservation of energy and conservation of momentum. Now, an algorithm could theoretically obey these laws and be incorrect, but in most cases energy and momentum analysis it is reliable.

This approach is also the most straightforward , since we simply take the output matrix of positions and calculate the total energy and moment at each time-step. The necessary equations for this are shown below:

$$E_{Total} = E_{Kinetic} + E_{Grav} \tag{2}$$

$$E_{Kinetic} = \frac{1}{2}mv^2 \tag{3}$$

$$E_{Grav} = -\frac{Gm_1m_2}{r} \tag{4}$$

$$\vec{p} = m\vec{v} \tag{5}$$

The correctness of the algorithm is given by how constant the total energy and total momentum are over the time-base. I have in this manner implemented a means of obtaining the energy and momentum over the time-base from the simulator. This can be used to assess the accuracy of a simulation.

4. DISCUSSION

Unfortunately this project ended up being more time-constrained than I would have hoped. I would like to have implemented additional summation and integration methods and to generally showcase them more. I also would have liked to include a comparison of methods in terms of both accuracy and performance.

As it stands, feel free to download the python files and play around with different starting conditions, in addition to a preset sample for the three-body figure-eight system described earlier. An example notebook should be sufficient to show how to operate the algorithm.

Overall, found delving into this topic very enjoyable as it is an interesting problem with some clever solutions. It is very satisfying to view animations of particles dancing around, having implemented the algorithm all from scratch. This has been a great learning experience for me, and it's been a great opportunity to apply my knowledge of both physics and computing to a problem that will likely have great relevance to my future career.

REFERENCES

2016, Three-body problem, Encyclopædia Britannica. https://www.britannica.com/science/three-body-problem

2019, , . https://cpb-us-e1.wpmucdn.com/sites.northwestern.edu/dist/2/77/files/2019/01/numerical_methods-1huaxg3.pdf

Barker, C. 2017, Numerical Methods for Solving Differential Equations, , . http://calculuslab.deltacollege.edu/ODE/7-C-1/7-C-1-h-c.html

Barnes, J., & Hut, P. 1986, Nature, 324, 446

Graps, A. 1996, N-Body / Particle Simulation Methods, , . http://www.cs.cmu.edu/afs/cs/academic/class/15850c-s96/www/nbody.html

Koby, T. ????, N-body Simulations, The Trustees of Princeton University. http://physics.princeton.edu/~fpretori/Nbody/intro.htm

McMillan, S. ????, The Leapfrog Integrator, , . http://www.physics.drexel.edu/~steve/Courses/Comp_Phys/Integrators/leapfrog/

Montgomery, R. 2010, Scholarpedia, 5, 10666, revision #137285

Springel, V., Yoshida, N., & White, S. D. 2001, New Astronomy, 6, 79 . http://www.sciencedirect.com/science/article/pii/S1384107601000422