# N-Body Simulation using the Barnes-Hut Algorithm

Shashanth Sriramanathan

May 2024

## 1 Introduction

This project explores the implementation of the Barnes-Hut algorithm in an n-body simulator. The Barnes-Hut algorithm is tree based and uses approximations to improve performance compared to a direct summation method. This allows the performance of the algorithm to scale as $\mathcal{O}(NlogN)$ rather $\mathcal{O}(N^2)$ (N is the number of particles) - a direct summation algorithm would scale as $\mathcal{O}(N^2)$, hence this is a significant improvement, especially when working with a large number of particles.

### 1.1 Initializing Particles

The initial positions of the particles were set using the Plummer Model

$$\rho(r) = \frac{3M}{4\pi a^3}\left(1 + \frac{r^2}{a^2}\right)^{-5/2} \tag{1}$$

which describes the density profile of stars in a stellar cluster. By working in N-body units, where $r_v = M = G = 1$ (where $r_V$ is the Virial radius, $M$ the total mass of the cluster and $G$ the gravitational constant), we can convert this into a CDF, then invert it to enable us to sample positions from the distribution:

$$F^{-1} = \frac{r_V}{\sqrt{p^{-2/3} - 1}} \tag{2}$$

where p is a probability.

The velocities of the particles have several configurations that were used. The most important is differential rotation. For differential rotation, the particle's velocity is determined using the total mass present within its radial position:

$$v(r) = \sqrt{\frac{GM(< r)}{r}} \tag{3}$$

## 2 Methodology

### 2.1 Creating the Tree

The tree is created recursively. We start with defining a 'root' node, which is essentially the box within which all the particles lie. The root node contains three properties; *bbox*, which defines the bounding box, *children*, which is a list of subnodes and a *center of mass*, which is defined by all the particles within the node.

For each node, we define an *insert* method, which inserts a given particle into a given node. This considers whether or not the node has already been sub-divided and reacts correspondingly. The particles are, again, inserted into the tree recursively, starting with the root node: the tree is created by inserting all the particles into the root node.

### 2.2 Computing Forces

The optimization that makes this algorithm have better performance comes from the way the forces are calculated. When a particle is far enough away from a given node, we approximate the force from all the forces from within that node using the center of mass of the node (to define distance from the particle) and the total mass of the particle.

Whether or not we use the approximation is based on a threshold value, $\theta$. If $L/d < \theta$ (where L is the width of bounding box of the node and d is as defined below), we use the approximation. If $L/d > \theta$, we repeat the process for each of the sub-nodes. We sum all the force contributions to end up with the net force.

d is defined as:

$$d = |\mathbf{r_{com}} - \mathbf{r_i}| \tag{4}$$

where $r_i$ is the position vector of the particle.

The force is calculated using the standard netwonian equation for gravitational forces:

$$F = \frac{GMm}{r^2} \tag{5}$$

### 2.3 Integration

The integration method used for this project is the velocity-verlet method. This is a time symmetric integration method and hence, ideally, would lead to the total energy within the system being conserved. Unfortunately, this does mean that we cannot use an adaptive timestep since it does not work with this method.

For each time-step, in order to calculate the force to determine the acceleration, we create the tree at the current and next timestep, since the velocity verlet method needs both in order to update the velocity. The force does not need to be calculated at the half time step since we updating just the position by half a time step, which just needs the current acceleration and acceleration.

### 2.4 Defining Errors

In order to get an idea of how accurate our algorithm is, we can compare our results with varying values of $\theta$ with the result where we set $\theta = 0$. I have chosen to do this by just comparing the differences in the force calculations. It can be defined as

$$Error^2 = \frac{(\Delta F)^2}{n} = \frac{1}{n}\sum_{i=0}^{n-1}\left((F^{\theta}_{x,i} - F^{0}_{x,i})^2 + (F^{\theta}_{y,i} - F^{0}_{y,i})^2\right) \tag{6}$$

where we are summing the squares of the differences in the x and y components of the force vector, over all $n$ particles.

## 3 Results

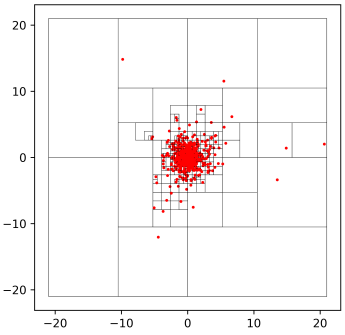
### 3.1 Example Results

In most of the simulations that were run, I noticed a large number of bodies that get ejected from the system, especially towards the beginning of the simulation. This also corresponded to large spikes in the total energy of the system.

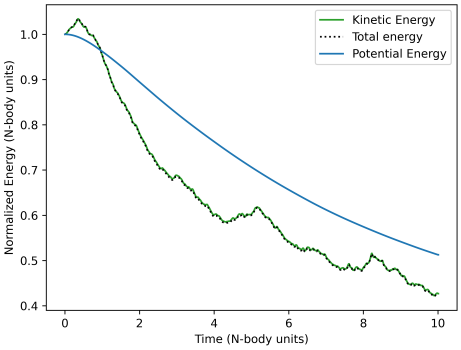
I was unable to create stable systems without a large central mass; all the runs with a normal/uniform distribution of masses resulted in only unstable systems.

### 3.1.1 1000 particles, no central mass

When initializing 1000 particles with differential rotation, a $\theta$ of 0.8 and no central mass (initial positions and the divided grid shown in fig 2a), there briefly seemed to have been a ring that formed (found in the folder *NoCentralMass_1000particles* in the github submission) that formed briefly before clumping up of the masses. The corresponding initial grid and evolution of energy are shown in fig 1. The reduction of energy with time is interesting because it is not a jump but rather a steady decline; I am not able to find a good explanation for why this is happening.



(a) Sub-divided grid of the initial positions of a 1000 particles



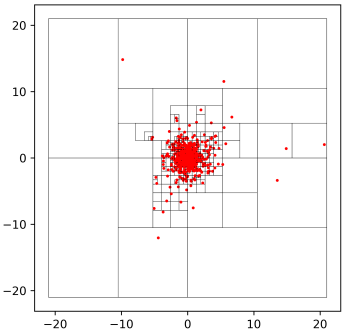

(b) Energy evolution of the system

Figure 1: System of a 1000 particles with no large central mass

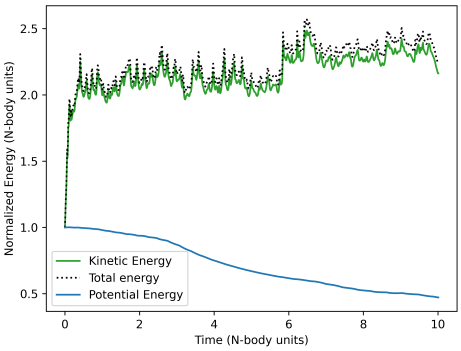
We also see that the total energy follows the kinetic energy fairly closely: this is because the magnitude of KE is a lot larger than that of the potential energy in the system.

### 3.1.2 1000 particles, 0.99 central mass

Simulating a 1000 particles for a maximum time of 10, with a central mass of 0.99 and initialized with differential rotation, resulted in a very stable system (apart from initial ejections) that very clearly looked like differential rotation. This is shown in the folder *LargeCentralMass_1000particles* in the github submission.



(a) Sub-divided grid of the initial positions of a 1000 particles




(b) Energy evolution of the system

Figure 2: System of a 1000 particles with a large central mass of M=0.99

## 3.2 Energy Conservation

Since we are using a velocity-verlet integrator. we expect the total energy in the system to be conserved (but also oscillate). An sample plot of energy against time is shown in 3. This

plot shows the total energy of the system (normalized by the energy in the initial state) for a simulation with a 1000 particles, run with $\theta = 0.8$, for a maximum time of 10 and central mass of 0.99. As we can see, for the most part, the total energy seems to be conserved

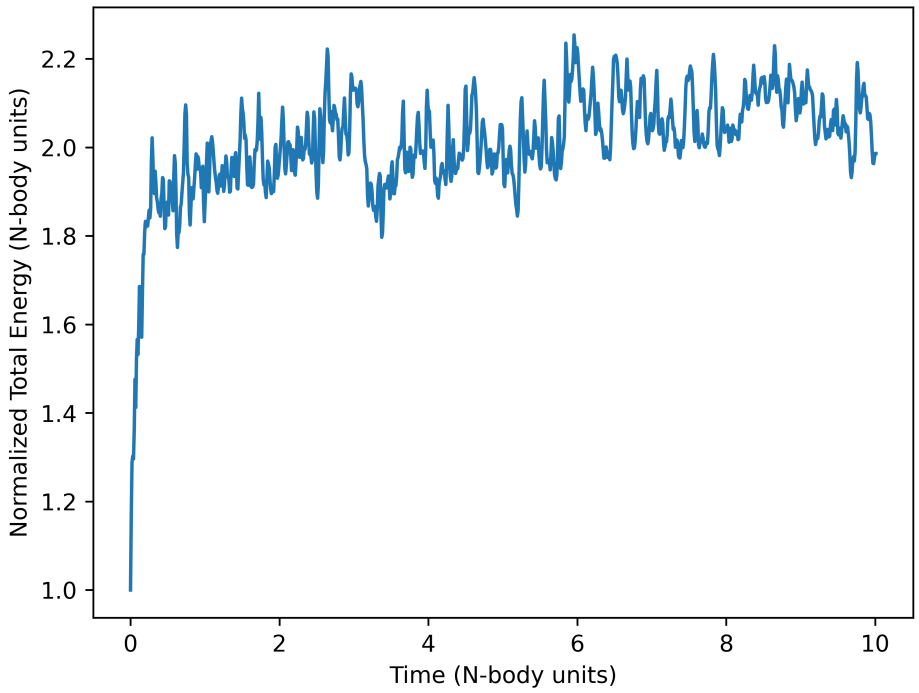



Figure 3: Evolution of the Total Energy in a system with a 1000 particles.

and we do see it oscillate. The jumps in the energy level correspond to close approaches between two objects, where our method does not have the required time resolution to work accurately. This causes a large amount of energy to artificially get injected into the system due to the large velocities of the particles at these instances.

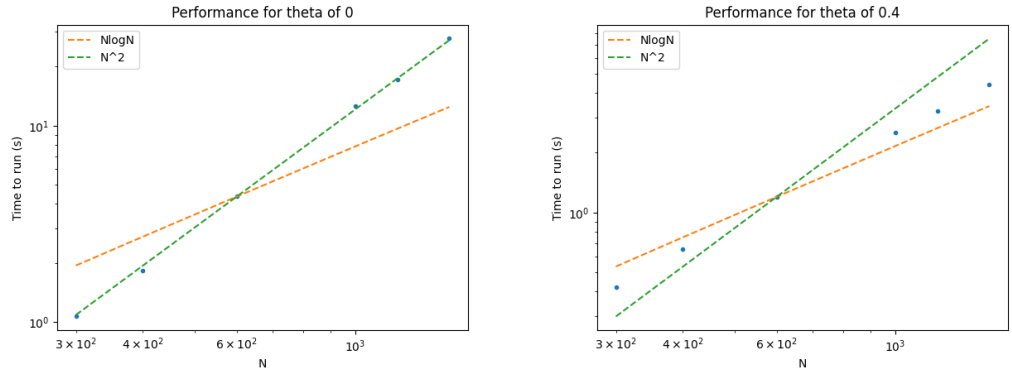
### 3.3 Performance

As mentioned earlier, we expect the performance of the Barnes-Hut algorithm to scale as $\mathcal{O}(NlogN)$. To test this, we time the algorithm for different values of N and plot its dependence on N. The plots are shown in figure 4 for different values of $\theta$.

We see that for $\theta = 0$, we have the same scaling as a direct summation algorithm, which is to be expected as for this value of theta we are essentially summing over all the particles. As we increase theta, we start aproaching the expected $\mathcal{O}(NlogN)$ scaling.

### 3.4 Errors

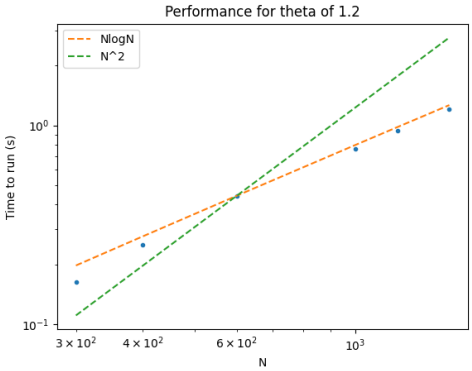
The relation of the error to $\theta$ is shown in fig 5 (the error is defined according to eqn 6). We can see that there is an expected trend upwards, exponential initially and flattening out for larger values of $\theta$, as we would expect. Interestingly, there also seems to discrete jumps at certain values of $\theta$.



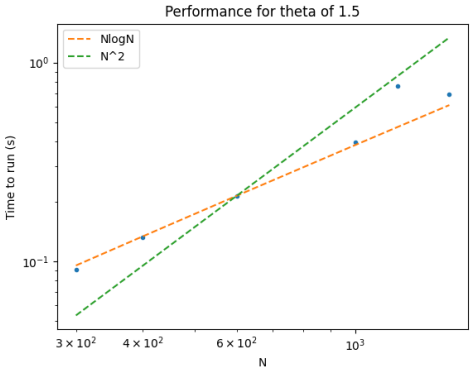

(a) $\theta = 0$




(b) $\theta = 0.4$




(c) $\theta = 1.2$




(d) $\theta = 1.5$

Figure 4: Scaling of Performance for different values of $\theta$

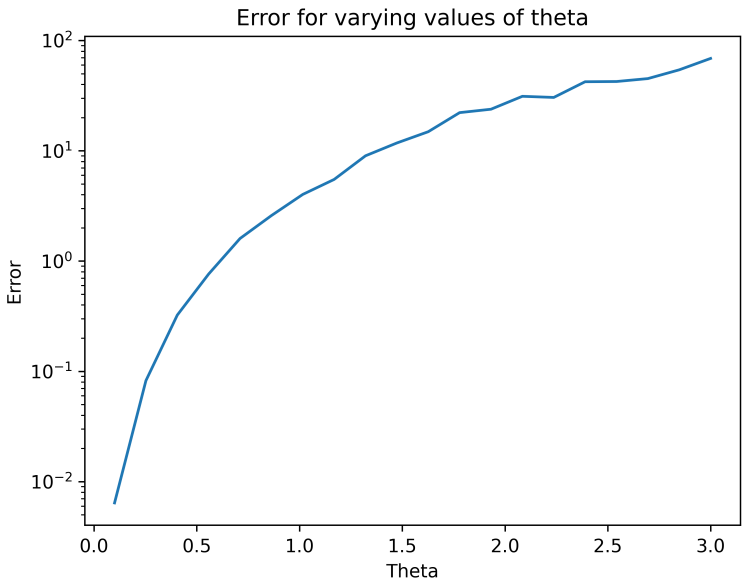



Figure 5: Dependance of the error on $\theta$

# 4 Remarks on the Algorithm

A possible way to improve the algorithm is by not re-creating the trees for each new time instant, instead we can check whether a particle has moved out of its current bounding box or not and update the tree correspondingly. This might allow us to re-use a for multiple iterations and hence end up saving time.

Another improvement can be done during the integration process. Since the velocity verlet method requires the acceleration at both the current and next time step in order to update a particle, we can save the force that we calculated at the next position and use that while performing the calculations for the next time step. This improves the efficiency as we now have to create the tree a significantly lower number of times.

As we increase the number of particles, the code takes longer and longer to run. This puts a constraint on the number of particles we can use based on the computing facilities we have access to and on how long we can physically run the code for. Another issue we might face is, if we have enough particles, we might get to a point where two particles are close enough that the computer thinks they are the same position. This would then lead to it reaching maximum recursion depth as we wont able to put the particles into two different nodes.