PHY407 2016 Final Computational Lab
Barnes-Hut Recursive Tree Algorithms

December 20, 2016

Contents

1 Introduction

N-body simulations are a powerful tool used in astrophysics (and many other areas of physics!) to model theoretically the behaviour of systems of N bodies. Without the use of a computer, we would have no hope of predicting the complex behaviour of say 10,000 particles analytically. Simulations on this complexity scale are computationally intensive, and in this report we investigate clever approximation algorithms that significantly reduce computational cost without great compromise to computational accuracy in systems evolving under Poissonian potential.

2 Physical Model

2.1 The Milky Way

In this lab, we will model the behaviour of a system of ~ 1000 particles under the influence of gravity in the hopes of reproducing galaxy-like behaviour, with the Milky Way in mind. The milky way is composed of matter distributed in a disk $\sim 15kpc$ in radius, and a dark matter halo $\sim 30kpc$ in radius. In reality the dark matter halo is extremely important, but in this simulation we will neglect contribution due to dark matter halo and perform the simulation on a 2D plane for ease of plotting. The Milky Way disk is composed of $\sim$ 1.5 trillion solar masses of matter. The sun is located at $\sim 8kpc$ from the core and has orbital period $\sim$ 200 million years. We would like to generate a initial system with similar conditions.

This motivates us to work in a system of units where length is in units of kpc, mass is in units of GMs (giga solar masses), and time is in units 10Myr (10 million years). In this system, G=0.449. Each of the 1000 particles will have mass $m = 1.5$, simulation box will have side length $L = 30$kpc, and average distance between particles will be $\epsilon \sim L/\sqrt{N} = 1$kpc.

2.2 Initial Positions

Matter in the Milky Way disk is distributed according to an exponential profile, where probability density P(R) of finding a star at radius R is given by Equation 2.1 (Nesti 2013 [1]), normalized over $R \in [0, \infty)$.

$$P(R) = (1/h_R)\exp(-R/h_R) \tag{2.1}$$

We can draw points $r \in [0, \infty)$ distributed according to Equation 2.1 by Monte Carlo importance sampling. We draw numbers $x \in [0, 1)$ uniformly and calculate r using Equation 2.2. We can give the point an arbitrary angular position by uniformly drawing from $\theta \in [0, 2\pi)$. This gives us a distribution of particles resembling the Milky Way shown in Figure 1.

$$r = -h_R \log(1 - x) \tag{2.2}$$

$$\hat{r} = \cos\theta\hat{x} + \sin\theta\hat{y} \tag{2.3}$$

2.3 Initial Velocities

Velocity curve in the Milky Way is well known from observational measurements. Theoretical predictions of velocity curve depend on whether dark matter is as shown in Figure 2. Naive estimate of velocity at radius R is $v(R) \sim \sqrt{GM_{gal}/R}$, orbital speed assuming concentration of galactic mass in the core (M_{gal} is mass

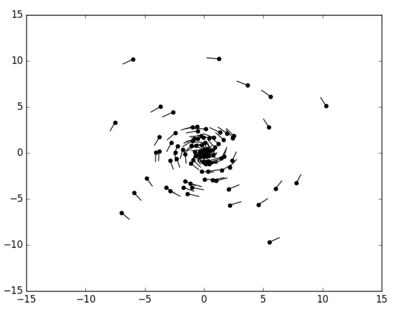
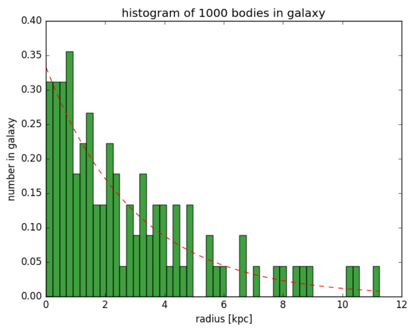
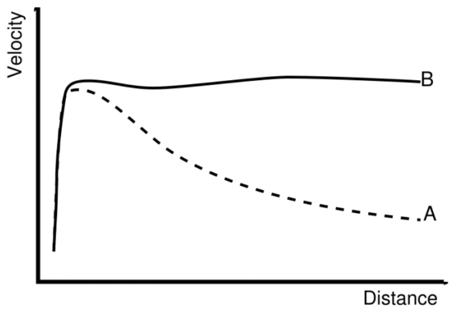



Figure 1: (left) 100 particles generated to resemble Milky Way galaxy. (right) Histogram of 1000 particles drawn by importance sampling over the interval $R \in [0, \infty)$. Dashed line plots Equation 2.1, the sampled distribution.



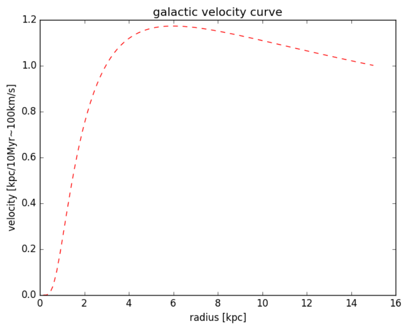


Figure 2: (left) Rotation curve for the milky where B accounts for dark matter and A does not (from Penn State Ast801). (right) Velocity function used to give speeds to particles at various R.

of galaxy). This matches behaviour of plot A for large R. We've implemented a boltzmann factor in Equation 2.4 that generates non-singular core behaviour as shown in Figure 2. We've generated a velocity distribution with direction perpendicular to radii using Equation 2.5 shown in Figure 1.

$$v(R) = \sqrt{GM_{gal}/R}\exp(-h_R/R) \tag{2.4}$$

$$\hat{v} = -\sin\theta\hat{x} + \cos\theta\hat{y} \tag{2.5}$$

2.4 Forces on System

The system evolves under the force of Gravity. Between every pair of particles, the gravitational force from particle 2 on particle 1 is given by Equation 3.2.

$$\vec{F} = \frac{Gm_1 m_2}{|\vec{r_2} - \vec{r_1}|^3}(\vec{r_2} - \vec{r_1}) \tag{2.6}$$

3 Computational Method

Below we describe basic theory of methods used as well as results of implementation. In the next section (Section), we will discuss detailed structure of Python implementation of these methods.

3.1 Computation of Force

Computation of force is the most computationally expensive part of the simulation. For each body, we must compute force from every other body. This makes $N(N-1)$ computations in total. Using this brute force method, we get complexity order $O(N^2)$. However this is slow and not all of it is necessary.

We begin by realizing that Poissonian potentials (like gravity) about a mass are very complex within some radius, but relatively smooth at large radii. This motivates us to try to approximate force due to masses at large distance with some kind of center of mass representative force rather than compute each force individually. This is the basis of the Barnes-Hut method (Barnes and Hut 1986 [2]), an order O(NlogN) method of computing forces in a system of N particles.

The basic idea is as follows. Particles are sorted into a quad tree, which is a recursive structure that divides a square recursively into smaller and smaller squares. Each square is a structure called a node, and each quadrant of the square is a child node. We keep dividing each quadrant of a square into subquadrants until every subquadrant has one and or zero particle in it. One that is achieved for any square, we stop. These squares are called leaf nodes, while each of the preceding squares are called branch nodes. Then all branch nodes have more than one particle in it, and thus must have child nodes. This procedure is implemented and the result is shown in Figure 3. Time taken to construct the tree is plotted against number of particles involved in Figure 3 where we can see that computational order is O(NlogN), which looks linear on long time scales. If this is not convincing, at least it is obviously not quadratic.

Every node of the tree is the same structure as its parent, so recursion is ideal for the construction of trees as well as accessing nodes. Python recursion however, is not tail optimized so it is very slow. This has the disadvantage of being slower than comparable O(NlogN) methods which rely on C programs like Particle Mesh which relies on FFT. In fact, it is around 25 times slower! This is

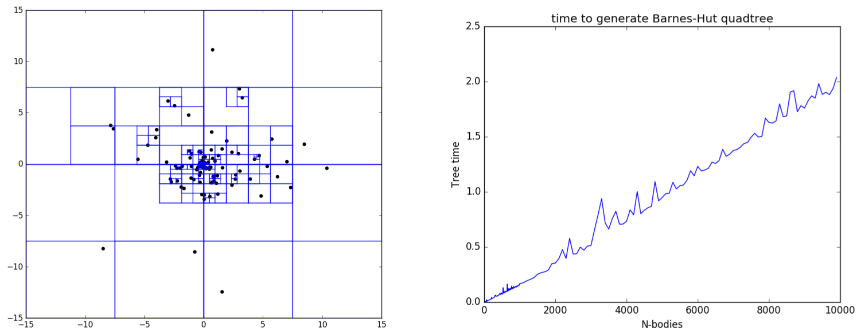


Figure 3: (left) Example construction of Barnes Hut Tree generated using **BHTree.py**. (right) Time required to construct tree as function of number of bodies (not that NlogN looks linear on long time scales).

shown by Battaglia (2016) who implemented the Particle Mesh code in parallel with this paper.

Each branch node of the tree stores only its child nodes (which are also trees in their own right), and a representative "aggregate body". The aggregate body is an object with mass equal to the sum of all masses contained in the quadrant represented by the node, and position equal to their centre of mass. It tracks this information easily because whenever a body is inserted into the node, it updates mass and position before inserting the body into the relevant child node. If the node is a leaf node, then it contains no children and its aggregate body is simply the one body it contains. Some children have no bodies and we should call them something else, like stub nodes.

To compute force on any given particle, we descend from the root node. At any node, if the aggregate body satisfies the inequality in Equation 3.1, then gravitational force from that node is added to force on that particle. Here s is size of node quadrant (side length), d is distance between particle and aggregate body, and θ_{BH} is a computational resolution parameter, where smaller theta increases resolution. We chose $\theta_{BH} = 1$ which means that resolution is fine enough when node body is at least one node length away. If the inequality is not satisfied, then we compute the gravitational force from the node on the body as the sum of the gravitational force from its child nodes. This operation completes in O(NlogN) time compared with $O(N^2)$ brute force algorithm as shown in Figure 4.

$$\frac{s}{d} < \theta_{BH} \tag{3.1}$$

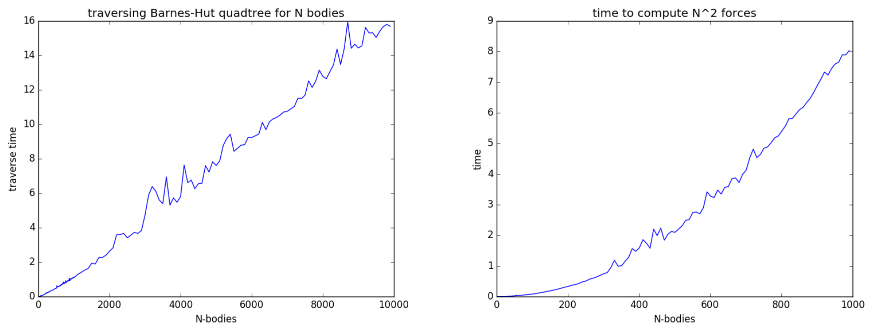


Figure 4: (left) Time required to evaluate forces on all particles from root node of Barnes-Hut tree as function of number of bodies generated using **BHtest.py**. (right) Time required to evaluate forces on all particles using brute force method as function of number of bodies generated using **BruteForce.py**.

3.2 Softening Length

A major problem encountered by all inverse square forces is that on finite timescales, should two particles take a step that takes them to within minuscule distance from each other, their force will be tremendous, maybe propelling them to infinity within a single time step. Energy conservation could go out the window. To prevent this, it is common practice to implement a softening length, that is the minimum length at which particles can interact. This modifies the force given in Equation 3.2 to Equation **??**, where ϵ is usually the average distance between particles (recall from Section 2).

$$\vec{F} = \frac{Gm_1m_2}{(|\vec{r_2} - \vec{r_1}| + \epsilon^2)^3}(\vec{r_2} - \vec{r_1}) \tag{3.2}$$

3.3 Integration of Motion ODE

The computational challenge is to calculate the force on every particle and integrate position and velocity in time. Suppose force F^i on every body were known at every point. The problem reduces to an ODE integration problem (Equation 3.4).

$$dv^i/dt = F^i(t) \tag{3.3}$$
$$dr^i/dt = v^i(t) \tag{3.4}$$

Leapfrog method would be ideal for simulations of systems under conservative forces (like gravity) for long time periods because energy is conserved. The discrete difference equations are given in Equation 3.7 (Newman 2012 [3]) with the first half step in velocity to start off the integrator shown. The error associated with this method is O(dt).

$$v^i(t_{1/2}) = v^i(t_0) + (dt/2) \times F^i(t_0) \tag{3.5}$$

$$v^i(t_{j+1/2}) = v^i(t_{j-1/2}) + dt \times F^i(t_j) \tag{3.6}$$

$$r^i(t_{j+1}) = r^i(t_j) + dt \times v^i(t_{j-1/2}) \tag{3.7}$$

Since Barnes-Hut simulation computes forces only on particles in the root node (30kpc box), we have implemented periodic boundary conditions to keep particles from escaping the simulation (and taking energy away). This is effected simply by Equation 3.8 which gives the root node the topology of a torus (or classic arcade game).

$$r = mod(r, 30kpc) \tag{3.8}$$

Barnes-Hut simulation of 100 bodies over box of 30kpc with initial consitions as described in 2 over time of 1Gyr with time step 1Myr conserves energy (Figure 5). Energy may fluctuate wildly locally, but on the long term energy is conserved to within 10%.

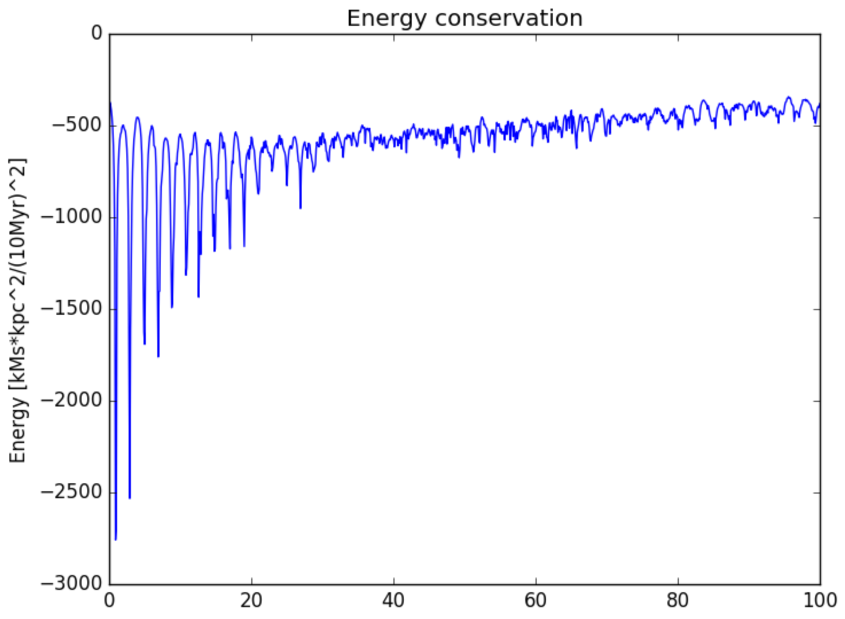


Figure 5: Conservation of Energy for 100 particles evolved using Barnes-Hut method over 1Gyr with time step 1Myr.

4 Python Implementation

An object oriented implementation of the methods described in Section 3 was written in Python.

BHSim.py is a script that runs the Barnes-Hut simulation. It has various user controllable parameters such as number of particles, radius of the simulation domain, mass of the galaxy, scale length of the galaxy (density drops to 1/e at scale length), Barnes-Hut resolution (θ_{BH}), softening length, time step, and total simulation time. The script produces an animation showing time evolution of the system.

Body.py defines the Body class, which is an object representing a physical particle in 2D space. It carries the particle's mass (float, units of GMs), position (2D array, units of kpc), velocity (2D array, units of kpc/10Myr), force (2D array, units of $GMs*kpc/(10Myr)^2$) box boundary (in kpc, for periodic boundary condition), and color (for plotting). Class carries functions for performing leapfrog steps (implementing equations in Equation 3.7), functions for computing distance between two body objects and checking position w.r.t. quadrants, functions for computing kinetic energy and potential energy, mutators for force on body, and a function for plotting.

Quad.py defines the Quad class, which is an object representing a quadrant in 2D space. It carries the position of its lower left corner (2D array), and its side length (float). Class carries functions that return Quad objects representing each of its subquadrants, and a function for plotting.

BHTree.py defines the BHTree class, which is an object representing a node of the Barnes-Hut tree. It carries a Quad object representing the quadrant that the node occupies. If it is not a stud node, it carries a Body object representing the aggregate particle in the node. If it has been given more than one particle, it carries 4 BHTree objects that are its child nodes. Class carries a function for inserting Bodies into the node. If it is a stub node, it accepts the Body and becomes a leaf node. If it is a leaf node, it changes its Body object into a aggregate particle, and gives its two Bodies to the appropriate children. If it is a branch node, it updates its aggregate Body and gives the new Body to its appropriate child. Class also carries a function for computing force of the node on a Body. If Body and the node's aggregate Body satisfy Equation 3.1, then force is computed between the two. If not, then force is computed from each of the node's children on the Body instead. If the node is a stub, then no force is added to the Body. Class also carries a function for recursively plotting the entire BHTree, with all non-stub children, grandchildren, etc.

MCgalaxy.py contains functions for generating various initial distributions of particles using Monte Carlo methods. It can generate a galaxy using methods described in Section 2, and it can also generate a uniform distribution of particles with random velocities.

BHtest.py is a script for testing speed and energy conservation of Barnes-Hut simulation.

BruteForce.py is a script for testing speed of $O(N^2)$ brute force simulation.

5 Results and Conclusion

The Barnes-Hut simulation seems to produce very physical results at much faster time than the brute force method. Comparison with Particle Mesh (Battaglia 2016) shows that it is much slower, but perhaps that can be improved by implementing recursion in C. An interesting observation in animations produced by this code is the tendency of galaxies to form circular shock waves that propagate out from the core. We believe it is the galaxy (initially not in equilibrium) virializing over time. Ring structure may be binary hardening ejecting particles at high speed while forming a low energy core. This is observable in nature in the form of galaxies like the Cartwheel galaxy which have recently been perturbed by an intergalactic interaction.

References

[1] F. Nesti and P. Salucci, "The dark matter halo of the milky way, ad 2013," *Journal of Cosmology and Astroparticle Physics* **2013** no. 07, (2013) 016. `http://stacks.iop.org/1475-7516/2013/i=07/a=016`.

[2] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," **324** (Dec., 1986) 446–449.

[3] M. Newman, *Computational Physics.* Createspace Independent Pub, 2012. `https://books.google.ca/books?id=SS6uNAEACAAJ`.