Barnes-Hut Algorithm for N-Body Simulations

Computational Statistical Physics - FS25 - ETH Zürich

Nicolas Müller
nicolmueller@ethz.ch

Abstract

This project report discusses a `C++` implementation of the Barnes-Hut algorithm for N-body simulations in 3D space. The implementation uses a custom octree data structure combined with the leapfrog time integration scheme. The algorithm is tested and verified using real world astronomical data from the NASA Horizons System and the ESA Gaia Archive. In a performance study it is then compared to the naive approach. Lastly, further potential improvements are suggested.

1 Introduction

The N-body problem is the problem of predicting the movement of a group of celestial bodies interacting through gravity. The classical two-body problem and the restricted three-body problem have been solved completely [Aar03]. For larger systems one has to resort to numerical simulations. Applications in astrophysics such as the the non-linear structure formation (such as galaxies) are an active field of research.

N-body simulations can be viewed as a particle simulation with gravity being the only interactive force between two particles i and j:

$$\vec{F}_{ij} = -G\frac{m_i m_j}{r^2}\hat{r}_{ji} \tag{1}$$

In a system of N bodies the number of pairwise interactions through gravitational force is $O(N^2)$. The naive brute-force approach to calculate all forces directly thus quickly becomes unfeasible for large N. This is where the Barnes-Hut algorithm comes into play.

2 Theory

The Barnes-Hut algorithm was introduced by Josh Barnes and Piet Hut [BH86] in 1986. The key idea of the algorithm is to group nearby bodies and approximate them through their center of mass (COM). This approximation can then be used to calculate the group's gravitational effect on other bodies that are far enough away. The COM of two bodies at positions (x_1, y_1) and (x_2, y_2) with masses m_1 and m_2 respectively is defined by

$$(x_{COM}, y_{COM}) = \left(\frac{x_1 m_1 + x_2 m_2}{m}, \frac{y_1 m_1 + y_2 m_2}{m}\right) \tag{2}$$

with $m = m_1 + m_2$.

The Barnes-Hut algorithm specifically groups and stores bodies in an octree (or quadtree in 2D) with the root node representing the entire simulation domain. Typically, every leaf node contains 0 or 1 body. This means that a node containing more than one body is subdivided into 8 children (or 4 children in 2D) . Each node stores its COM and total mass, i.e. that of all bodies contained in itself and its children. Figure 1 shows a simple

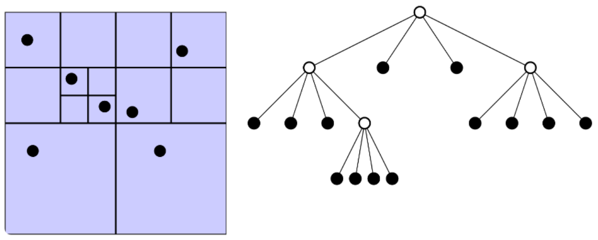

Figure 1: Illustration of a quadtree. Image source: [Rei]

visualization of a quadtree for a 2D application. Note the imbalance of the tree as nodes are only subdivided if multiple bodies are contained in the area covered by that node.

To then calculate the force acting on a specific body a recursive procedure starts at the root of the tree: If the COM of a node is sufficiently far away, the entire node (including all its children) is approximated as a single body. Otherwise, if the COM of the node is close, we continue this process recursively with the node's children. Once a leaf node is reached, the force interaction with the body stored in the leaf is computed directly.

The condition whether a node is far enough away is usually determined through a ratio $\frac{s}{d}$ of the distance d between the body and the node's COM and the size s of the node which is then compared to a threshold value θ. If $\frac{s}{d} < \theta$ the internal node is sufficiently far away and can be approximated by its COM. The parameter θ controls the speed-accuracy trade-off. Small values for θ improve accuracy. In the extreme case of $\theta = 0$ the algorithm degenerates to its brute force form where every body is treated directly and no approximations are used. A commonly used value is $\theta = 0.5$.

This algorithm effectively reduces the time complexity of the N-body simulation from $O(N^2)$ to $O(N \log N)$.

3 Implementation

The C++ implementation together with usage instructions and input data can be found in the ETHZ GitLab [Müla] or in a public GitHub repository [Mülb]. The following section gives a general overview of the implementation, for details please refer to the code itself which is carefully commented.

3.1 Octree

The heart of the software is, naturally, the octree itself. This data structure is implemented with a single `Node` struct:

```
struct Node {
    Vec3 position;
    Vec3 size;
    std::vector<Node*> children;
    std::vector<Particle*> particles;
    Vec3 center_of_mass;
    double mass;
};
```

Note that, contrary to the typical design of a Barnes-Hut octree, this structure allows for multiple particles per leaf node. The number is controlled through the `MAX_PARTICLES` macro. This design choice makes the tree more flexible at the cost of a small overhead induced by the usage of vectors.

To construct the tree, all particles are inserted one after the other at the root node. Once a node contains more than `MAX_PARTICLES`, it is subdivided into 8 children and its particles redistributed among them. Since only leaf nodes are allowed to contain particles, all internal nodes (including the root) pass particles down recursively when they are inserted. When passing down a particle `p`, nodes update their total mass and COM. This can be done very efficiently with the following code snippet (also see equation 2):

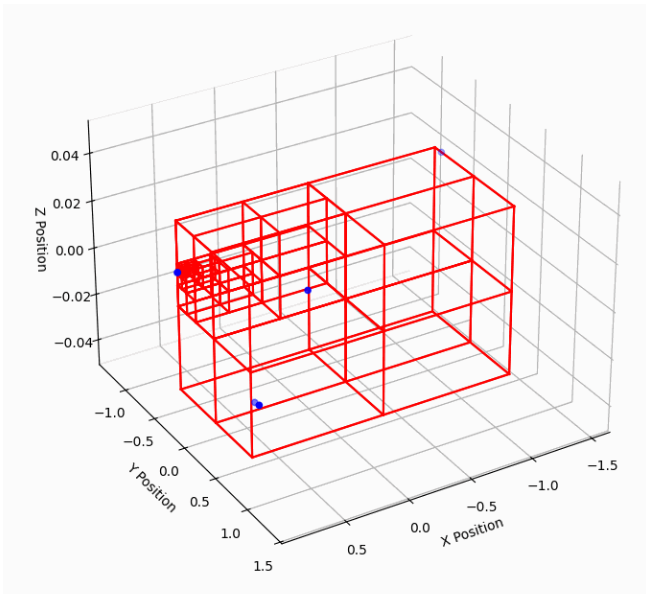

Figure 2: Visualization of constructed octree for a small test ensemble of 5 particles

```
double total_mass = mass + p->mass;
center_of_mass = (center_of_mass * mass + p->position * p->mass) / total_mass;
mass = total_mass;
```

The tree is built from scratch at every timestep. Updating an existing tree after the bodies update their positions would not only be a much more complex task, but often less efficient than simply rebuilding it. Figure 2 shows a visualization of the implemented octree data structure.

The algorithm that describes the walk along the tree during force calculation was already discussed in section 2. Its implementation is straight forward as the description translates directly into code. Therefore, it is not explored here any further.

3.2 Time integration

For the time integration of the equations of motion, the most common scheme for many astrophysical N-body simulations is the leapfrog method. Being a symplectic integrator it ensures energy conservation over long time periods. At the same time it is rather simple and efficient to implement. Leapfrog updates the velocity in half steps, requiring two force calculations per time step and therefore also two octree constructions per time step.

3.3 Units

An important aspect that needs to be consiered to avoid problems with numerical precision is the choice of units. Since this application is operating with astronomical data a suitable choice is astronomical units (1 AU = $1.495 \times 10^{11} m$), years and solar masses ($1 M_\odot = 1.9884 \times 10^{30} kg$). Consequently, the gravitational constant in units of $\frac{\mathrm{AU}^3}{M_\odot \mathrm{yr}^2}$ is given by $G = 4\pi^2$.

Astronomical units and years are a good option when simulating systems similar in size compared to our solar system. When going beyond that (e.g. see section 4.2) one should consider parsecs (pc, $3.0857 \times 10^{16} m$) and meagyears (Myr).

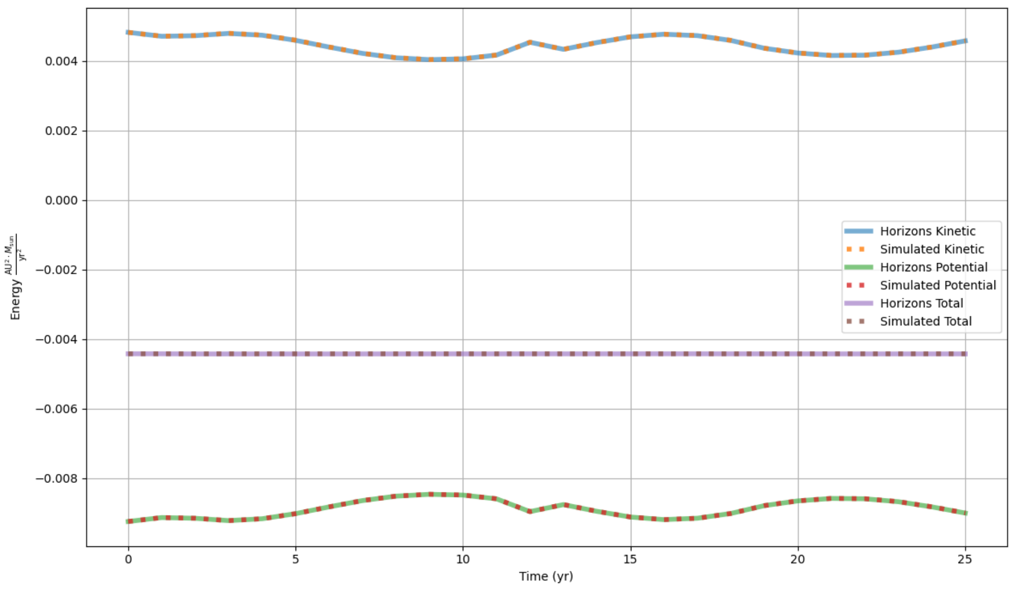

Figure 3: Potential, kinetic and total energy of the solar system over time. Comparison between simulation output and reference data. Energy is measured in $\frac{\mathrm{AU}^2 M_\odot}{\mathrm{yr}^2}$.

4 Results

4.1 Correctness

To verify correctness of the implementation, the publicly available JPL Horizons [NAS] data set was used which i.a. provides access to positions and velocities of the planets in our solar system. Of course, the eight planets together with our sun only form a very small data set which does not benefit from the scaling effects of the Barnes-Hut algorithm discussed in section 2. However, it is a convenient and illustrative way to ensure that the results from our force calculation and time-integration are consistent with reference data.

The Horizons data is queried directly with a Python script using the *astroquery* library. As discussed in section 3 we need to ensure sensible units. In this case astronomical units (AU), years (yr) and solar masses ($M_\odot$). The system state is initialized with the Horizons data from January 1st 2000. It is then integrated for 25 years in time steps of $\Delta t = 1$day. Every year the simulation state is compared to the Horizons reference data.

Figure 3 shows the potential, kinetic and total energy for both the simulated and reference data. All three align very closely.

To check whether the individual positions of each planet are also consistent, Figure 4 shows the absolute difference between the simulated position and the reference position. Deviations up to 0.08 AU are observed. Since the JPL Horizons data accounts for effects far beyond Newtonian gravity (relativistic corrections, planetary oblateness, perturbations from many minor bodies, etc.) which are not part of this implementation, this level of error seems acceptable.

It is further noted, that the position difference over time shows a very similar structure for all bodies. This hints at a general system wide drift or similar perturbation. The integration itself however, appears to be very stable within the chosen time frame.

4.2 Performance

To show the benefits of the Barnes-Hut implementation over the naive approach, a much larger data set is needed. One possibility would be to just randomly place particles in a simulation box since, for a pure performance study, the bodies do not necessarily need to follow a stable orbit. However, in real world scenarios, the distribution of celestial bodies in space and thus the structure of the Barnes-Hut octree might not follow a uniform distribution. Therefore, to get a fair estimate for common application scenarios, real world data is also

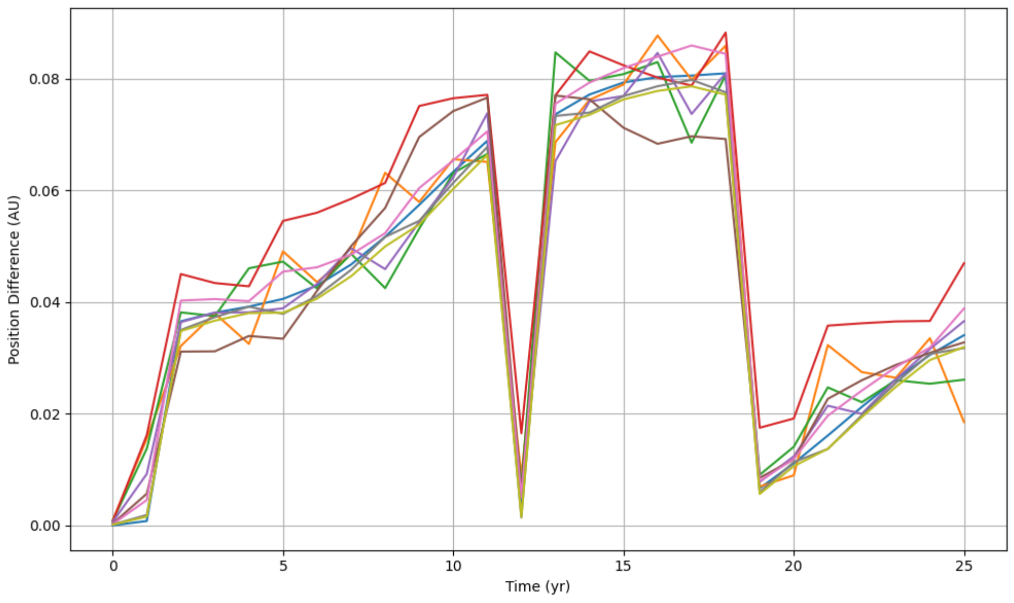

Figure 4: Absolute difference between simulated planet positions and reference data over time

preferable for a performance study.

For this, a set of 50 thousand stars are queried from the Gaia ESA Archive [ESA]. The cluster is shown in Figure 5. The change in scale of this new system requires another rescaling of our units to avoid numerical problems. Switching from astronomical units (AU) to parsecs (pc) and recalculating the gravitational constant G accordingly ensures stability.

To compare the Barnes-Hut algorithm to the naive approach the runtime is plotted for an increasing number of particles in Figure 6. For small counts (in the lower hundreds) the octree implementation is slower than the naive approach due to the overhead of constructing and traversing the tree. However, it quickly surpasses the naive approach for larger counts. When simulating 50'000 particles it is over 130 times faster with $0.144s$ per step compared to $18.89s$.

Allowing multiple particles per leaf, in this case 8, improves performance further. This observation has to be treated carefully as the specific tree implementation in this project is intentionally designed to allow a flexible number of particles per leaf by (see section 3). However, the usage of vectors becomes an unnecessary overhead when only storing one particle per leaf.

N	**Naive (s)**	**Octree (s)**	**Octree8 (s)**
100	0.000094	0.000200	0.000097
1000	0.007914	0.002702	0.000905
10000	0.787255	0.027668	0.008441
49992	18.894040	0.143599	0.044441

Table 1: Direct comparison of average runtime (in seconds) for different implementations and particle counts N.

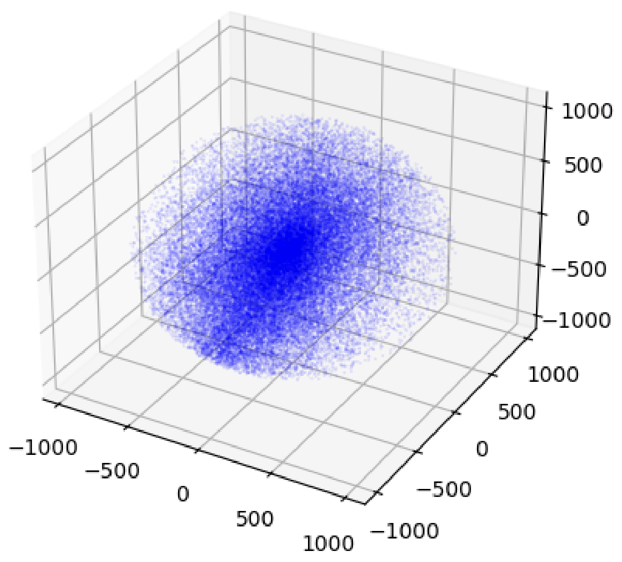

Figure 5: Scatter plot of 50′000 stars from the Gaia Archive used for performance analysis. Length scale in parsecs (pc).

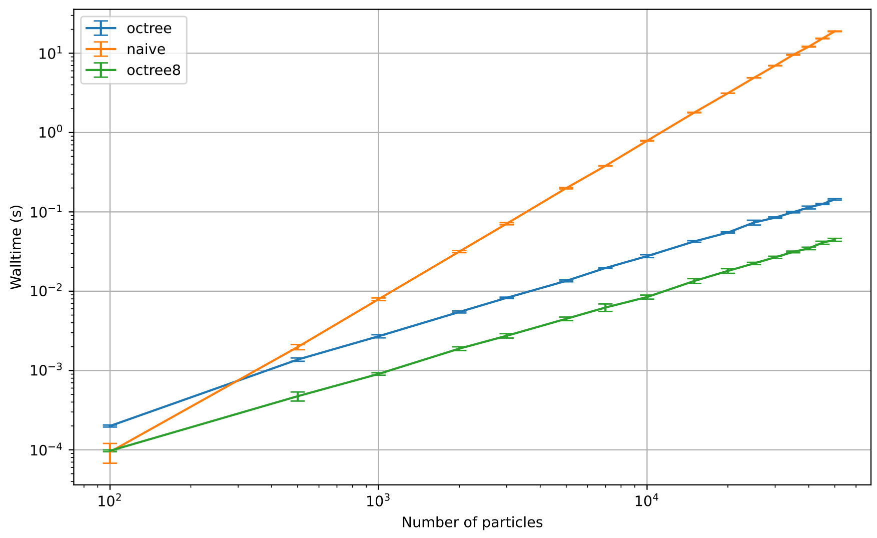

Figure 6: Performance study for the octree implementation with one particle per leaf (octree) and 8 particles per leaf (octree8) compared to the naive implementation (naive). The walltime (in seconds) of a single leapfrog step is measured for 100 up to 49993 particles. Each data point represents the arithmetic mean of $n = 10$ measurements, the error bars represent the standard deviation. An additional warm up step is performed whenever the particle count is changed to account for memory effects. All measurements were performed on a single Intel Core i5-12600K at 3700 MHz.

5 Conclusion and outlook

In this report it was shown that the implementation is working correctly and can reproduce observed data with reasonable accuracy. The performance scaling for increasing number of bodies behaves as expected and outclasses the naive approach by orders of magnitude in the systems studied.

Future improvements to the implementation could include adaptive time step control based on maximum accelerations or an iterative instead of recursive approach to force calculation. Force calculation and tree walk could also be parallelized using OpenMP rather easily while parallel tree construction remains a very difficult task. Furthermore, a memory pool for node allocation could be used to optimize memory management by reducing the repeated creation and destruction of nodes and reusing them instead. Minor improvements to cache optimization could probably be achieved by using fixed-sized arrays instead of `std::vector`s inside the `Node` struct. Lastly, this report has not covered parameter optimization regarding θ. Larger values would further speed up computation at the cost of accuracy.

References

Aarseth, Sverre J. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 2003. ISBN: 9780521432726.

Barnes, J. and P. Hut. "A hierarchical $O(N \log N)$ force-calculation algorithm". In: *Nature* 324.6096 (Dec. 1986), pp. 446–449. DOI: 10.1038/324446a0.

ESA. *Gaia Archive*. URL: https://gea.esac.esa.int/archive/.

Müller, Nicolas. *GitHub Project Repository*. URL: https://github.com/MuellerNico/barnes-hut.

— *GitLab Course Repository*. URL: https://gitlab.ethz.ch/nicolmueller/csp/-/tree/main/termproject.

NASA. *Horizons System*. URL: https://ssd.jpl.nasa.gov/horizons/.

Reinarz, Anne. URL: https://annereinarz.github.io/slides/barneshut/content.html.

Declaration of originality

I hereby affirm that I have independently produced this paper and the associated code, and adhered to the general practice of source citation in this subject-area.

Generative AI has been used in the context of this project to assist with code generation (primarily GitHub Copilot and GPT4o). AI answers were cross-checked with reliable sources whenever appropriate. Generative AI has **not** been used for any text generation in this document.