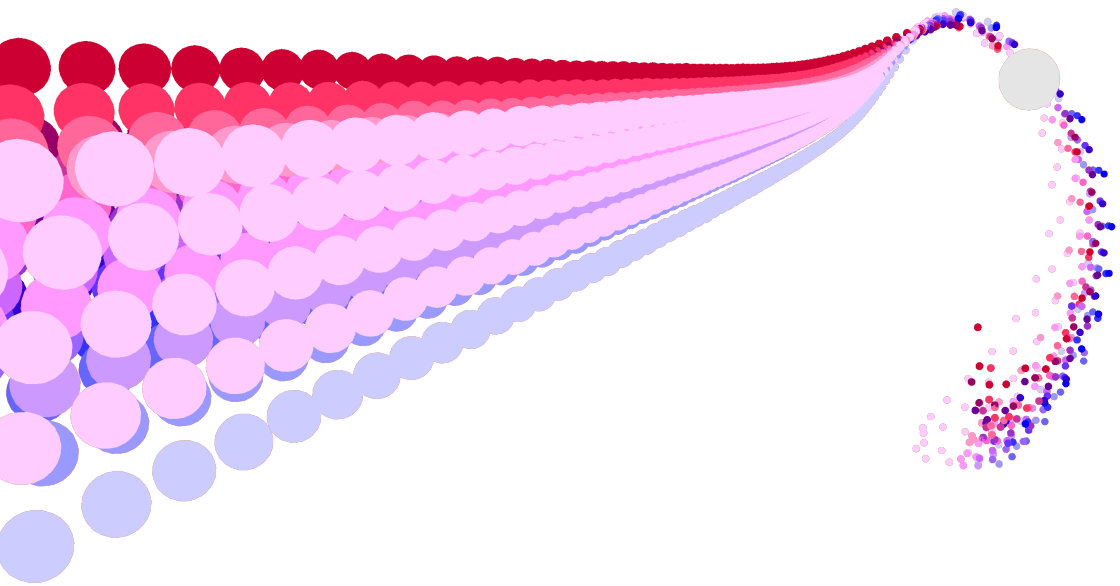
Dual Hierarchy *for* Gravitational n-body

Version of August 17, 2023



Jackson Campolattaro

Dual Hierarchy *for* Gravitational n-body

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Jackson Campolattaro
born in Columbia MD, USA

Computer Graphics and Visualization Group
Department of Intelligent Systems
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: *A scenario containing 12,501 point-masses simulated using the Dual Hierarchy algorithm, rendered as circles.*

Dual Hierarchy *for* Gravitational *n*-body

Author: Jackson Campolattaro
Student id: 5625270
Email: j.r.campolattaro@student.tudelft.nl

Abstract

The *n*-body problem is the simulation of pair-wise interactions between *n* objects. This problem appears in many forms, with the classic example being the modeling of gravitational forces between point masses, necessary for cosmological simulations. Many approximation approaches have been devised to reduce the complexity of this problem.

t-SNE is a data visualization method that requires repeatedly solving a variant of the *n*-body problem. A recent paper (van de Ruit et al. [2022]) proposes a novel algorithm that outperforms other t-SNE minimization methods on medium-scale datasets. The report proves the viability of a dual-traversal method that uses an embedding tree to emit forces and an independent field tree to collect forces. Because the embedding tree is a Linear-BVH and the field tree is an orthtree built to a fixed depth, the overall algorithm has linear complexity.

This thesis demonstrates how the dual-tree approach can be adapted for gravitational *n*-body simulations. Following this, it measures the performance against similar implementations of other algorithms and shows that while the adapted Dual Hierarchy approach is faster than Barnes-Hut, it is outperformed by the Fast Multipole Method on realistic large-scale cosmological datasets.

Thesis Committee:

Chair:	Prof. Dr. E. Eisemann, Faculty EEMCS, TU Delft
University supervisor:	Dr. K. Hildebrandt, Faculty EEMCS, TU Delft
Committee Member:	Dr. M. Taouil, Faculty EEMCS, TU Delft

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The n-body problem is the name for the task of predicting the motion of n particles, all of which apply forces to one another. Variants of the n-body problem appear throughout classical physics. At very small scales, plasma physics, molecular dynamics, and protein folding must be modeled iteratively by calculating the electrostatic forces between particles. At the far end of the spectrum, planetary systems, galaxies, and galactic clusters can only be modeled by integrating the forces of gravity between celestial bodies over time.

Of these problems, the gravitational variant has received the most attention. Modeling the motion of stars and galaxies is crucial for understanding the early universe. Techniques ranging from the Fourier Particle-Mesh approach to a host of tree-codes have come out of research into faster cosmological simulations, and have been subsequently applied in other fields.

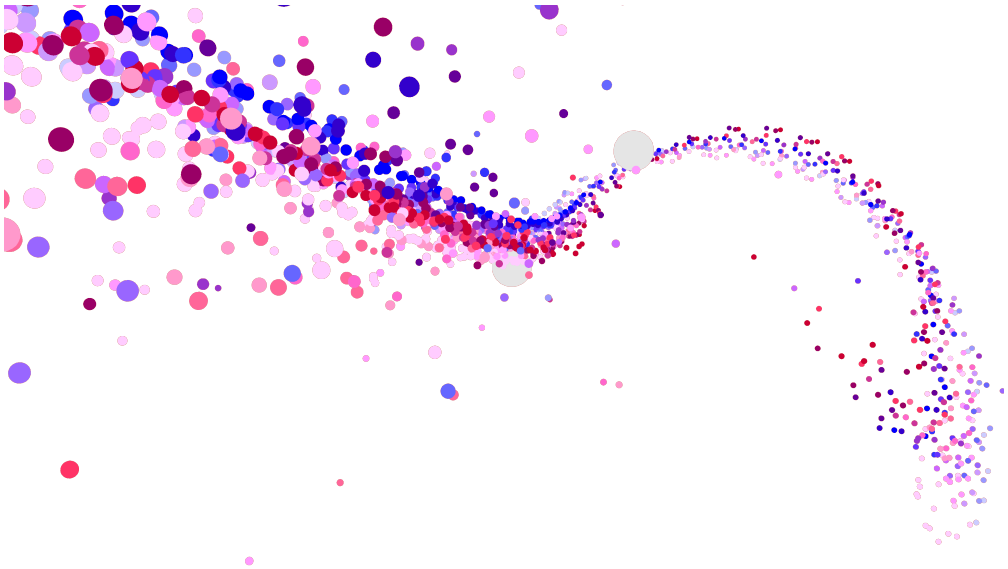


Figure 1.1: A synthetic gravitational n-body scenario containing 25,000 low-mass particles stirred by a binary pair of heavy "star" masses.

Another formulation of the n-body problem appears in t-SNE minimization. t-SNE is a visualization technique that produces a locality-preserving embedding of

a high-dimensional dataset in 2 or 3 dimensions. Samples in the high-dimensional dataset are assigned probabilities that they belong to the same group, based on their relative proximity. In the lower-dimensional space, samples start with a randomized arrangement and are iteratively improved until a high-quality embedding is produced. We improve the positions by simulating attractive forces between pairs of particles with high probabilities, and repulsive forces between all others.

The attractive forces in t-SNE minimization can be calculated efficiently separately, but calculating the repulsive forces requires solving the n-body problem. This is done using algorithms such as Barnes-Hut adapted from gravitational n-body, as well as algorithms such as Particle-Mesh, from the plasma physics variant of the problem. Other algorithms have since been designed which take advantage of the reduced accuracy requirements of t-SNE.

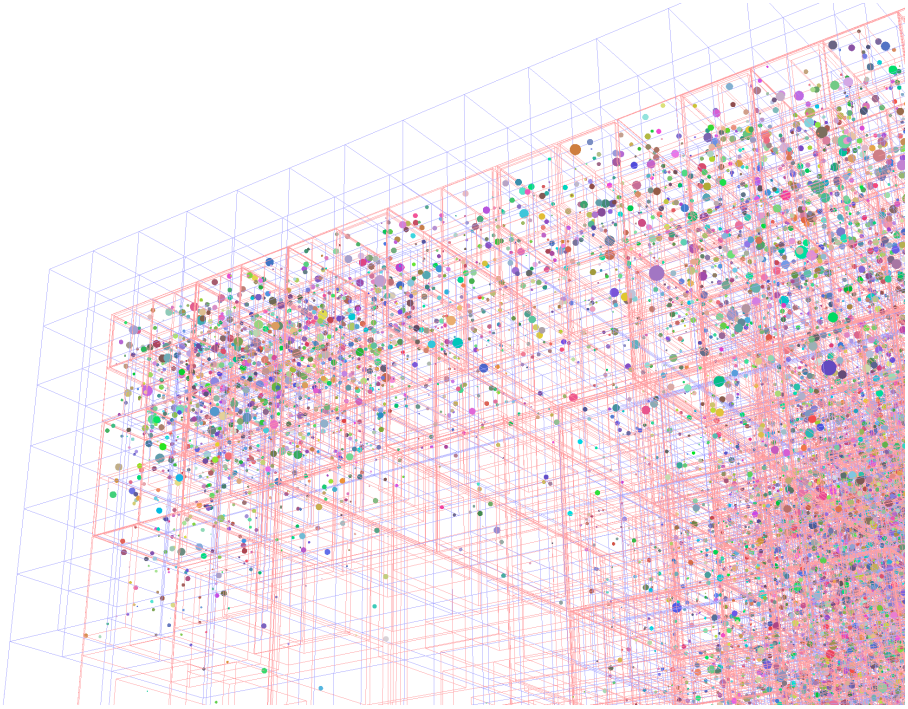


Figure 1.2: A Linear-BVH (Red) and an Octree (Blue) partitioning a collection of particles in 3D space, randomly distributed using Perlin noise.

The focus of this report is on one such algorithm, proposed in the paper van de Ruit et al. [2022]. This paper describes a Dual Hierarchy method for calculating the repulsive component of the t-SNE minimization force. The defining feature of the Dual Hierarchy algorithm is its use of two separate trees. A Linear-BVH is constructed to represent the force-emitting side of the process, and a Quadtree with fixed depth is constructed to receive forces. In this way, a coarse field can be efficiently computed in linear time. Figure 1.2 shows how the two hierarchies partition the same space.

Unlike other t-SNE algorithms, the Dual Hierarchy method has no major features which prevent it from being adapted to gravitational n-body. In this paper, we demonstrate a working adaptation and optimize it for the new domain. In Chapter 4, we explain each of the changes which need to be made to the algorithm in order to achieve

the increased accuracy requirements of cosmological simulations. We also explore changes to the algorithm which can improve its performance in the new domain. The modified algorithm degenerates to particle-particle interactions to ensure the necessary accuracy is achieved at very close ranges and uses quadrupole moments and octupole fields to improve the accuracy of node-node and node-particle interactions at longer ranges.

The Dual Hierarchy method comes with several advantages in the new domain: A Linear-BVH is generally higher quality than an Octree; more balanced, and with no singleton chains. The binary splitting of the tree provides a greater variety of node sizes, meaning more opportunities for approximation. Most important are the tight-fitting bounding boxes of Linear-BVH nodes, which capture more information about the particles they contain. A Linear-BVH node will generally be smaller than that of an octree for the same distribution of contained points, so we can perform more approximations for a given value of θ. Moreover, because we can use the center of an Linear-BVH node to reason about distance, we can make more accurate accuracy guarantees with fewer edge cases (Section 4.2.3), this allows for higher values of θ.

Chapter 2 summarizes relevant research into the optimization of Gravitational *n*-body and t-SNE, respectively. Together with Chapter 3 it provides the background information necessary to understand the remainder of this report. Chapter 4 details the approach used to adapt Dual Hierarchy to the new problem domain, and describes several optimizations applied to the resulting implementation. In Chapter 5 we compare the adapted Dual Tree algorithm to other state-of-the-art algorithms used for gravitational *n*-body. We show that while the novel algorithm outperforms Barnes-Hut, it underperforms the Fast Multipole Method due to the advantages inherent to that algorithm. In Chapter 6 we conclude by recommending the adaptation of the Fast Multipole Method for use in t-SNE. We show that several factors which had led to the initial dismissal of dual-traversal for t-SNE can be eliminated by combining features from the Fast Multipole Method and Dual Hierarchy for t-SNE.

Chapter 2

Related Work

In Section 2.1 we discuss the current state of gravitational n-body research and a variety of algorithms now in use. In Section 2.2 we discuss several methods for t-SNE minimization, most of which only have a single implementation.

2.1 Gravitational n-body

The algorithms used for gravitational n-body have been constrained by the need for high accuracy combined with the need to perform well on datasets with extremely high dynamic ranges. These factors make treecodes a natural fit, and the algorithms which are used in contemporary gravitational n-body projects generally descend from the Barnes-Hut method.

a) The Particle-Particle, Particle-Mesh (P^3M) method originally proposed in Eastwood et al. [1980] was the first major improvement over the naive solution. It adapts the Particle-Mesh (grid-field) approach used for contemporary plasma simulations. Far-field interactions between nodes in the grid are computed efficiently in Fourier space. Near-field interactions are computed directly between particles in order to achieve the necessary accuracy. This provided a significant speedup over the $O(n^2)$ naive approach, but because it degenerates to the naive approach in areas of high density, it ultimately has the same complexity.

b) The original Barnes-Hut method was introduced in Barnes and Hut [1986]; it provides an approximation which reduces the complexity of gravitational n-body from $O(n^2)$ to $O(n\log(n))$. In the Barnes-Hut method, an octree subdivides the point masses, and long-range forces are computed using node-particle interactions, where nodes are represented by their center of mass. The exact functioning of Barnes-Hut is explained in more detail in Section 3.2.1. This paper has had a major influence on the direction of the field and remains relevant to this day. This is in part due to the simplicity of the method, which has made it straightforwardly adaptable to other variants of the n-body problem.

Barnes-Hut has been optimized significantly since its original proposal. It is used in solvers for small systems such as Rebound (Rein and Liu [2012]), where quadrupole moments are used to produce more accurate results. It remains relevant as the near-field component of larger solvers such as ChaNGa (Jetley et al. [2008]), which imple-

ments Barnes-Hut using a hybrid approach that traverses local sections of the tree on the GPU (Jetley et al. [2010]).

c) The Fast Multipole Method (FMM) was introduced in Ambrosiano et al. [1988]. FMM improves on Barnes-Hut in two key ways: First, a dual-traversal is performed over the octree, and node-node interactions are used wherever possible. This reduces complexity from $O(n\log(n))$ to $O(n\log(\theta))$, effectively $O(n)$ for fixed values of theta. Second, multipole representations are used to more accurately summarize the masses within a node on the force-emitting side of the traversal, and more accurately summarize the acceleration over a node on the force-receiving side. This enables the use of lower values of θ than would otherwise be possible. The details of FMM and its differences with Barnes-Hut are explained in Section 3.2.2.

d) Tree Particle-Mesh (TreePM) was first proposed by Bagla [2002]. The algorithm is a hybrid of Barnes-Hut and the older P^3M. As in P^3M, Particle-Mesh is used to find long-range interactions in Fourier space. Barnes-Hut is used in place of the naive approach to compute near-field interactions accurately. The overall complexity of TreePM is still $O(n\log(n))$, but in practice, it outperforms the FMM approach even for very large values of n. Importantly, partitioning the simulation domain into cells makes the algorithm naturally amenable to distributed computing implementations. Most large-scale simulations use periodic boundary conditions, and this is typically done using Ewald summation, which approximates these very long-range forces using the first several terms of an infinite sum. Because TreePM computes long-range forces in the frequency domain, these periodic boundary conditions can be incorporated without additional computation.

e) Fast Multipole Method Particle-Mesh (FMM-PM) is proposed by Springel et al. [2021], and implemented by the GADGET-4 project. It is conceptually similar to TreePM, but uses the dual-sided FMM traversal for middle-distance interactions in place of Barnes-Hut. This project also improves performance and load balancing on high-dynamic-range "extreme zoom" simulations by using higher-resolution Particle-Mesh grids in areas of higher density. Because GADGET-4 is CPU-bound, it does not categorically outperform the ChaNGa project.

The majority of recent gravitational n-body research has focused on non-algorithmic improvements to the accuracy and speed of the overall simulation.

The GADGET-4 project (Springel et al. [2021]) takes into account a variety of physical effects beyond gravity. These range from the fluid mechanics of gaseous matter to star formation and evolution, with each update adding more detail. They are incorporated into the system without requiring changes to the underlying treecode.

A major advancement in recent years is the use of enhanced time-dilation techniques. A simulation can be finished in less time despite using the same force calculation algorithm if the time steps can be larger. Multistepping approaches use finer-grained time-steps for particles under high acceleration, and update the accelerations of other particles less frequently. Some form of adaptive time stepping is incorporated into most projects, but the PDKGRAV3 project (Potter et al. [2016]) takes the concept much further, using a hierarchy of time steps, where each level is calculated half as frequently as the last. For "extreme-zoom" simulations with very high dynamic range, they use 100 levels, meaning that the particles experiencing the greatest acceleration are updated $2^{100}\times$ as often as those experiencing the least acceleration.

Other projects have focused on making better use of available hardware. For example, Ishiyama et al. [2022] implements the TreePM method for the supercomputer Fukagu, and makes optimizations to inter-node communication in order to scale effectively to millions of threads.

Crucially, these advances do not place specific requirements on the algorithm used for force calculations. An improved n-body treecode would be compatible with additional physics calculations and with enhanced time-dilation. An improved treecode could also be integrated into algorithms that use Particle-Mesh for long-range forces, replacing the Barnes-Hut algorithm in TreePM, or the FMM approach used in FMM-PM.

2.2 t-SNE

t-SNE has been the subject of optimization efforts for less time than gravitational n-body, with the technique first proposed in Maaten and Hinton [2008] and tree algorithms not finding use until Van Der Maaten [2014]. The variant of the n-body problem solved as part of t-SNE minimization has several differences from the gravitational variant, which have affected the direction of research:

a) t-SNE is typically applied for datasets of the sort used in machine learning. These can be extremely large, but the n-body minimization step works on the embedding, where each sample is represented by a coordinate in 2- or 3-dimensional space, with no attached mass or other properties. Not all samples must be included in the embedding to achieve high-quality results, and a large-scale t-SNE minimization may only have n on the order of millions. As a result, there has been a focus on optimizing t-SNE minimizers for a single computer, rather than a large cluster.

b) t-SNE has significantly reduced accuracy requirements versus gravitational n-body. Because t-SNE is an iterative minimization problem, some error in individual force calculations is allowable so long as it does not prevent the eventual convergence of the embedding. As the algorithm converges on a solution, separately computed attractive forces dominate in the near-field, further relaxing the accuracy requirements of repulsive forces. Multipole methods and other approaches to increase accuracy have not been applied, and several schemes which are not viable for gravitational n-body can be applied for t-SNE minimization.

c) Because t-SNE is a stochastic algorithm, the minimization step may be run many times with different initial embeddings before an acceptable result is produced. An acceptable result is only identifiable by hand, so a person must decide whether to re-run the minimization. Because of this, latency has been prioritized over throughput. Momentum systems help the embedding converge in fewer iterations, and implementations have focused on reducing memory transfers to and from the GPU. The hybrid CPU-GPU schemes which now dominate gravitational n-body have not seen the same success for t-SNE.

Some algorithms which work well for gravitational n-body were quickly applied to t-SNE.

a) The use of Barnes-Hut for t-SNE was originally proposed in Van Der Maaten [2014]. Barnes-Hut was later adapted to run entirely on the GPU in Chan et al. [2018],

and this approach was subsequently improved to use an implicit tree structure and make better use of the GPU in Meyer et al. [2020], and Meyer et al. [2021].

b) Van Der Maaten [2014] also explores an approach that performs a dual-traversal on a single octree or quadtree, analogous to a simplified version of FMM. This approach was dismissed at the time because it underperformed the implementation of Barnes-Hut from the same report in both speed and accuracy.

Other algorithms have only been possible for t-SNE because of unique properties of the problem domain, and have no direct analogues in gravitational n-body.

a) The Linear-complexity t-SNE approach proposed by Pezzotti et al. [2020] takes advantage of GPU circuitry meant for textures, and uses it to compute a field texture containing rasterized forces. For each sample in the embedding, a repulsive force texture is rendered to the overall field texture using additive blending. Naturally, the algorithm only works in 2-dimensions, but in exchange for this limitation, it can outperform Chan et al. [2018] for small-scale datasets, allowing for interactive performance in some cases. Besides only working in 2 dimensions, this algorithm is not applicable for gravitational n-body because of the accuracy penalty of rasterization.

b) The main advancement in Fu et al. [2019] is an improved method for setting embedding starting conditions, but it also demonstrates a different approach for minimization. This approach estimates the net force on each embedding point by randomly sampling non-neighbor points. This enables eventual convergence, as all repulsive forces are included in at least some steps, and can reduce runtime by a factor of 2 over Chan et al. [2018] in certain cases. Because this algorithm compromises the accuracy of individual iterations, it cannot be used for gravitational n-body.

c) Linderman et al. [2019] proposes Fast Interpolation-based t-SNE (FIt-SNE), which computes accelerations over a quantized field in Fourier space. This is similar to the Particle-Mesh approach, but it does not use a more accurate method for near-field interactions in the way that P^3M or TreePM do. This allows for a 10 times speedup over an implementation of Barnes-Hut t-SNE, and operations at much larger scales, where memory limitations become an issue for algorithms such as Chan et al. [2018]. As with Linear-t-SNE, the accuracy penalty of the quantized field means the algorithm cannot be transferred to gravitational n-body.

The Dual Hierarchy strategy proposed by van de Ruit et al. [2022] performs dual-traversal in much the same way as FMM. The paper shows that an optimized version of the algorithm can outperform the state-of-the-art in t-SNE minimization for medium- to large-scale datasets. It can produce minimizations of equivalent quality in less time than optimized versions of Barnes-Hut and FIt-SNE at small and medium scales. The functioning of the algorithm is described in detail in Section 3.3.

Chapter 3

Background

This section provides the necessary information for understanding the original Dual Hierarchy algorithm, the challenges involved in its adaptation to gravitational n-body, and the technical details of the adaptation's implementation.

Section 3.1 gives a more formal definition of the n-body problem as it appears in cosmological simulations and in t-SNE. Section 3.2 relevant existing algorithms for gravitational n-body, and Section 3.3 explains the Dual Hierarchy algorithm as it is used for t-SNE. Section 3.4 details some of the metrics used to evaluate the accuracy of cosmological n-body simulations, and Section 3.5 explains Multipoles and how they are used to improve the accuracy of these simulations.

3.1 The n-body Problem

The n-body problem typically refers to the gravitational n-body problem, but it is also the name for an entire class of problems involving the computation of forces between every pair in a set of n points in space. For the purposes of this report, we will use the latter definition. Typically, the magnitude of a force must be calculated to be applied along the vector connecting each pair. The n-body problem appears frequently in physics simulations, where the forces between particles decrease with distance, but never go to zero. Simulating Coulomb interactions between ions or gravitational interactions between stars both involve solving an n-body problem, but calculating collision forces between particles does not qualify, as there are no long-range interactions.

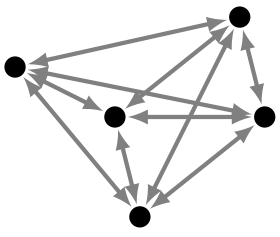


Figure 3.1: Solving the n-body problem requires computing forces between each pair.

3.1.1 Gravitational n-body

Astronomical simulations can range from small simulations of planetary systems to galactic simulations containing many stars, all the way to cosmological simulations which examine galactic cluster formation. The most expensive step in running these simulations is predicting the movement of the point masses, as driven by gravity. This can be done analytically for two bodies, and for special cases with more than two bodies (Aarseth [2003]), but astronomical simulations require solving the general case, with $n \gg 2$. For this, it is necessary to predict the movement iteratively, solving the n-body problem at each step to find the forces between point masses.

Algorithm 1 A naive approach for n-body gravity.

```
1 for each [p_passive, a_passive] ∈ particles do
2     a_passive ← ⟨0,0,0⟩
3     for each [p_active, m_active] ∈ particles do
4         a_passive ← a_passive + GRAVITY(p_active, m_active, p_passive)
5 for each [v_passive, p_passive] ∈ particles do                ▷ Integration step
6     v_passive ← v_passive + (a_passive · t)
7     p_passive ← p_passive + (v_passive · t)
```

In the gravitational variant of the n-body problem, each point-mass is typically represented by a scalar mass value m and a Cartesian coordinate in 3-D space p. The naive approach to the gravitational n-body problem involves calculating the acceleration due to gravity between each pair of particles, as shown in Algorithm 1. In this part of the algorithm $p_{passive}$ is the position of the particle which is receiving forces, and $a_{passive}$ is its acceleration. p_{active} is the position of the particle emitting forces, and m_{active} is its mass. GRAVITY() is a function that computes the acceleration at the position of the passive particle due to the mass and position of the active particle. Because acceleration is force over mass, the mass of the passive particle cancels out, and we can calculate its acceleration using only the mass of the active particle.

The second half of the algorithm updates first each velocity $v_{passive}$ and then each position $p_{passive}$ based on the time step t. In Algorithm 1, this is done using simple Newtonian integration, but in practice, leapfrog integration or a more accurate symplectic approach is typically used. Time subdivision techniques are also typically incorporated as part of the integrator.

t-SNE updates positions using a different scheme, and the original Dual Traversal paper does not propose any new integration methods which would be relevant to gravitational n-body. The integration step is common between all n-body solvers discussed, so it will be omitted elsewhere.

3.1.2 n-body for t-SNE Minimization

Student-t Stochastic Neighbor Embedding (t-SNE) is a technique for visualizing high-dimensional data in a 2- or 3-dimensional space. t-SNE works on collections of high-dimensional samples; for example, MNIST handwritten digits are 28×28 pixels, but we can reinterpret each pixel as a dimension, meaning that the dataset is 784-dimensional. The goal is to produce a neighbor-preserving embedding, meaning that

samples that were close together in the high-dimensional space (e.g. images with similar pixel values) are also nearby in the embedding. An example of such an embedding is shown in Fig. 3.2.

Figure 3.2: A completed embedding of the MNIST digits dataset, produced using t-SNE; the color of each sample corresponds to the digit. For example, each dark blue point corresponds to a single handwritten "2".

This goal is defined more formally as the minimization of the sum of Kullback-Leibler divergence between the original particle distribution and that of the embedding. This gives us a cost function C expressed as follows:

$$C = \sum_i KL(P \mid Q) = \sum_i \sum_j p_{ij} \log(\frac{p_{ij}}{q_{ij}}) \tag{3.1}$$

Where p_{ij} and q_{ij} are the symmetrized conditional probabilities that data points i and j belong to the same group in the original dataset (P) and the embedding (Q). A Gaussian function is used to define p_{ij}, but because distances are longer in the higher ambient dimension of the original space than in the embedding, data points with low probabilities would be placed very far apart if a Gaussian was used on both sides. Instead, a Student-t distribution is used for the joint probabilities of the embedding because its "fatter tails" mean that dissimilar data points still have some probability of belonging to the same group:

$$q_{ij} = \frac{\sum_{k \neq l} (1 + \|y_k - y_l\|^2}{1 + \|y_i - y_j\|^2} \tag{3.2}$$

Where y_i is the position of data point i in the embedding space. Given these definitions for the conditional probabilities, the gradient of the cost function is derived as:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j \frac{(p_{ij} - q_{ij})(y_i - y_j)}{1 + \|y_i - y_j\|^2} \tag{3.3}$$

In practice, the Gaussian probability is near zero for all but a handful of data points. We can take advantage of this by only calculating p_{ij} for the nearest data points j to

data point i, with the number of probabilities to calculate determined by a hyperparameter "perplexity". With p_{ij} zero for most elements, it makes sense to refactor the gradient equation:

$$\frac{\delta C}{\delta y_i} = 4\sum_j p_{ij}\frac{(y_i - y_j)}{1+\|y_i - y_j\|^2} - 4\sum_j q_{ij}\frac{(y_i - y_j)}{1+\|y_i - y_j\|^2} \tag{3.4}$$

The left half of the equation can be calculated quickly using the exact formula, as it is zero for all but a small number of data points j. The right half must be evaluated for all j, and this is an instance of the n-body problem. This is the focus of most t-SNE optimization efforts.

Once we have an efficient method for computing gradients, we can use gradient descent to find an embedding that minimizes the Kullback-Leibler divergence. We assign each sample a random starting position in the lower dimensional space. From there, we perform an error minimization, iteratively improving the positions until we converge on a stable embedding. With our split gradient function, this is analogous to applying attractive forces between a small number of similar particles while applying repulsive forces between all pairs of particles.

t-SNE Accuracy Metrics

The objective of t-SNE is to minimize the Kullback-Leibler divergence between the original dataset and its embedding. Directly computing this is on the order of $O(n^3)$, so measuring accuracy is prohibitively expensive for even relatively small datasets. Instead, the original Maaten and Hinton [2008] paper focuses on the perceived quality of the embedding, which primarily means the cluster should have clusters that cleanly map to the classes of a known dataset. This is ultimately more important than the ability to satisfy the objective function, as it determines the usefulness of the algorithm. The subjective measure is supplemented by a test of the effectiveness of a nearest-neighbors classifier trained on the embedding; for a high-quality embedding, its performance is similar to that of one trained on the original dataset.

Meyer et al. [2022] proposes a new computable proxy for visual quality which provides a global measure of quality rather than only looking at local neighborhoods. This is important because the attractive component which dominates local behavior is computed exactly in most t-SNE implementations, so local neighborhoods may not be representative of overall quality. These metrics are still based only on the quality of the embedding produced at the end of convergence. The accuracy of individual steps and gradient calculations only matters insofar as it reduces the number of steps needed to converge on a solution.

3.2 Tree Methods

In most variants of the n-body problem, the derivative of the pairwise force has greater magnitude at short distances. This means that it is less sensitive to positional error at longer distances. We can take advantage of this by computing interactions with far-away clusters, rather than finding the pairwise forces individually. This significantly reduces the number of interactions that must be computed as long as we have an accurate approach for summarizing these clusters.

It makes sense to produce our cluster summaries ahead of time. We can use very large clusters for the longest-range forces, but we need to have progressively smaller clusters for closer interactions. A multigrid can work in many situations, but for simulations with very high dynamic range – areas of much lower and higher density – a tree makes the most sense. Each node of the tree contains some subset of the particles in the simulation. The children of an internal node each contain a subset of the particles in that node.

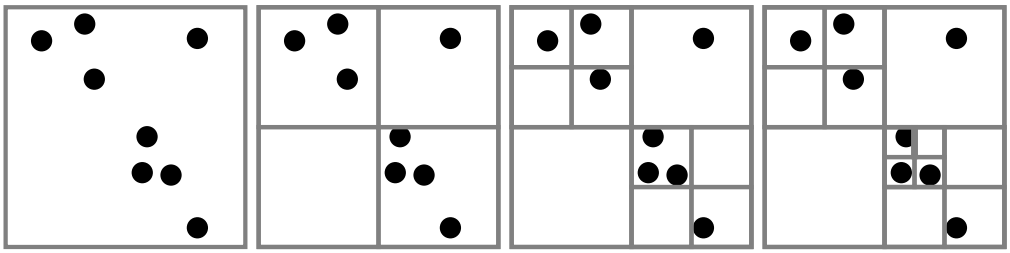
To compute the interactions, we descend the tree. For each node, we check if it is far and small enough to provide an accurate approximation. If we cannot interact directly with that node, we try each of its children instead. If we reach a leaf node but it is too large, we may interact with all of the contained particles individually. In this way, we account for every particle in the tree, either directly or as part of a summary.

Tree methods exist for many domains. The two most commonly used treecodes for gravitational n-body are Barnes-Hut, discussed in Section 3.2.1, and FMM, discussed in Section 3.2.2.

3.2.1 The Barnes-Hut Method

The Barnes-Hut Algorithm is useful as an explanatory tool due to its simplicity, but also performs well enough to remain relevant today.

In Barnes-Hut, the particles are partitioned using an octree in 3D space, or a quadtree in 2D space, as shown in Fig. 3.3. Each node is summarized with a center of mass, denoted by the symbol ◌ in Fig. 3.4. This can be done efficiently because higher-level summaries can be produced from those of lower-level nodes. At the limit, interaction with a center of mass is equivalent to the combined interaction with all contained particles. At ranges closer than infinity, the distribution of the contained particles affects the interaction force. At close ranges, this error is unbounded. We can use the center of mass for interactions with nodes that are sufficiently far.



(a) Root node contains all particles. (b) Partition particles around the center. (c) Over-filled nodes are partitioned again. (d) Repeat until all particles are separate.

Figure 3.3: Top-down construction of a Quadtree from a collection of particles

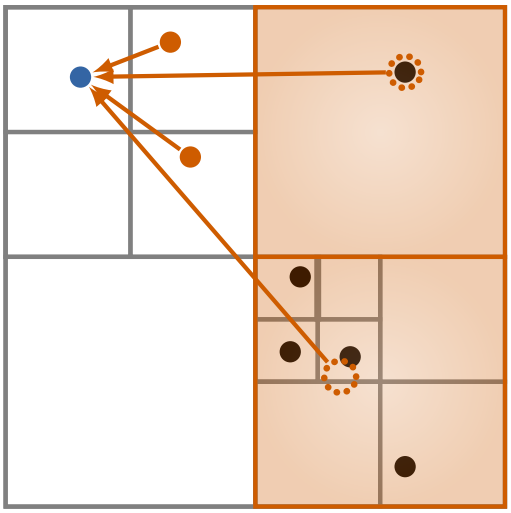


Figure 3.4: In Barnes-Hut, close-range forces are computed directly with particles, and long-range forces use the centers-of-mass of nodes.

To decide whether a node is sufficiently far, the concept of a Descent Criterion or Opening Criterion is introduced. In the case of Barnes-Hut, the descent is based on a fractional angle θ, as shown in Eq. (3.5).

$$\frac{S}{D} < \theta \tag{3.5}$$

Where S is the side length of a node, and D is the distance between the sample point and the center of mass of the node. This criterion produces a "cone of approximation", which includes small nodes nearby and larger nodes further away, as shown in Fig. 3.5.

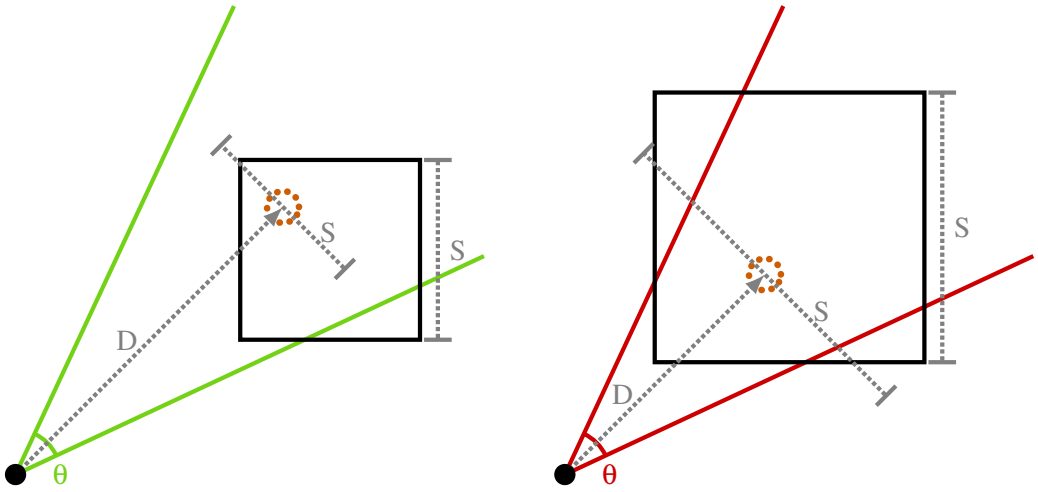


Figure 3.5: An example of a node which satisfies the θ descent criterion (left), and a node which does not (right).

Algorithm 2 outlines an approach to computing the approximate net acceleration due to gravity a at a point in space $p_{passive}$ due to a tree node and its contents. If the

node is a suitable approximation as determined by the descent criterion, the acceleration is found by calculating the gravitational field of the center of mass of the node at $p_{passive}$. Otherwise, the tree is recursively descended. In cases where a leaf node is reached and no approximation is possible, the algorithm degenerates to direct interactions. This happens for very close interactions, such as those between particles in the same node.

Algorithm 2 A recursive implementation of Active-Tree Traversal

```
1  procedure TRAVERSE_ACTIVE(node, p_passive)
2      a ← (0,0,0)
3      if DESCENT_CRITERION(node, p_passive) then
4          a ← a + GRAVITY(p_active, m_active, p_passive)
5      else if IS_LEAF(node) then
6          for each [p_active, m_active] ∈ node do
7              a ← a + GRAVITY(p_active, m_active, p_passive)
8      else
9          for each child ∈ node do
10             a ← a + TRAVERSE_ACTIVE(child, p_passive)
       return a
```

This algorithm describes all hierarchical approaches which use an "active tree", meaning a hierarchy that summarizes the point masses for the purpose of emitting forces. When the descent criterion is given by Eq. (3.6) and the node is part of an octree, this is equivalent to the Barnes-Hut method.

$$\text{DESCENT\_CRITERION}(\text{node}, p_{passive}) = \begin{cases} \textbf{True} & \frac{S_{node}}{\|p_{node} - p_{passive}\|} < \theta \\ \textbf{False} & \text{Otherwise} \end{cases} \tag{3.6}$$

We perform the traversal for the position $p_{passive}$ of each particle, as shown in Algorithm 3. So long as we can guarantee no pathological cases, where many particles occupy the same space, tree construction is done in $O(n\log(n))$ time. Each of n traversals is done in $O(\log(n))$ time, so the complete algorithm has $O(n\log(n))$ complexity.

Algorithm 3 Application of Barnes-Hut traversal to find particle accelerations

```
1  root ← BUILD_OCTREE(particles)
2  for each [p_passive, a_passive] ∈ particles do
3      a_passive ← TRAVERSE(root, p_passive)
```

To expand this to a multi-threaded context, we can assign each thread a collection of particles for which it finds forces. Because traversal does not modify the active tree, it can be shared between all threads without locking. Tree construction can also be done in parallel, though it requires additional care. The first several layers of the tree are constructed in one thread, and then the subtrees can all be constructed independently.

3.2.2 The Fast Multipole Method

The Fast Multipole Method (FMM) was first proposed in Greengard and Rokhlin [1987], and improved by using Cartesian coordinates in place of spherical coordinates

in Dehnen [2000]. By traversing both sides of the tree and performing node-node interactions, complexity is reduced to $O(n\log(\theta))$. For fixed θ, this is equivalent to linear time. Aluru [1996] showed that when applying the algorithm with fixed accuracy requirements, the required θ is coupled with n, meaning that the effective complexity is actually super-linear.

To perform a node-node interaction, we need to calculate an approximation of the field across the "passive" node which receives forces as a result of some summary of the mass distribution in the "active" node which emits them. The obvious approach would be to use the center-of-mass representation for the active node as with Barnes-Hut, and evaluate the acceleration at the center-of-mass of the passive node. Unfortunately, this is extremely inaccurate – even at very long ranges, the acceleration due to gravity will not be uniform across a node. This makes it impossible to meet the accuracy requirements of cosmological n-body simulations without using a value of θ so low that the benefits of the tree algorithm are lost.

The solution is to use multipole approximations. The center-of-mass summary is replaced with a multipole moment, and the uniform acceleration summary is replaced with a multipole field. Multipoles are explained in more detail in Section 3.5, but to understand FMM it is only necessary to know that a multipole moment is a compact way of approximately describing the distribution of a collection of point masses, and a multipole field is a compact way of describing the variation across a non-uniform vector field.

An FMM long-range interaction is shown in Fig. 3.6. In the diagram, ✡ denotes a multipole moment, and ✳ denotes a multipole field representation. Notice how the moment is at the center of mass of the active node, but the field is sampled at the center of the passive node. This is because the field multipole is placed in order to minimize the maximum error across the node, meaning that it must be equidistant from all sides.

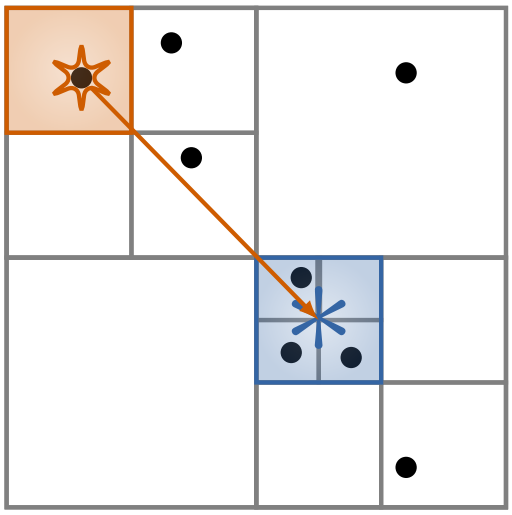


Figure 3.6: In FMM, long-range forces can be calculated between an active node (orange) and a passive node (blue).

As with Barnes-Hut, we use a θ descent criterion to determine whether a node-node approximation can be used, or if the tree must be further descended. A simple implementation of this concept is shown in Algorithm 4. We refer to this approach

as *lockstep* dual traversal because if the approximation criterion is not satisfied, *both* nodes are descended. When we reach a leaf node on either side of the tree, we decay to either active-tree traversal (equivalent to Barnes-Hut), or passive-tree traversal (equivalent to reverse-Barnes-Hut, explored in Section 4.1.2). In practice, passive-tree traversal is very slow, so most implementations prioritize the use of active-tree instead. More elaborate approaches are explored in Section 4.4.1.

Note how if both nodes are leaves, an active tree traversal is used. For most of the particles contained by the passive node, this will decay immediately into direct interactions. Some of the particles on the far side of the passive node will interact with the entire active node instead, thereby saving some computation as compared to directly decaying to particle-particle interactions.

Algorithm 4 A recursive implementation of Lockstep Dual Traversal

```
1  procedure TRAVERSE_LOCKSTEP(node_active, node_passive)
2      if DESCENT_CRITERION(node_active, node_passive) then
3          [p_passive_node, a_passive_node] := node_passive
4          [p_active_node, m_active_node] := node_active
5          a_passive_node ← a_passive_node + GRAVITY(p_active_node, m_active_node, p_passive_node)
6      else if IS_LEAF(node_passive) then
7          for each [p_active, m_active] ∈ node_passive do
8              a_passive ← a_passive + TRAVERSE_ACTIVE(node_active, p_passive)
9      else if IS_LEAF(node_active) then
10         for each [p_passive, a_passive] ∈ node_active do
11             TRAVERSE_PASSIVE(p_active, m_active, node_passive)
12     else
13         for each child_passive ∈ node_passive do
14             for each child_active ∈ node_active do
15                 TRAVERSE_LOCKSTEP(child_active, child_passive)
```

3.3 The Dual Hierarchy Algorithm for t-SNE

The primary novel characteristic of the Dual Hierarchy algorithm is its use of two different trees, one which is responsible for emitting forces, and another which is responsible for receiving them.

In the original implementation, the first tree is referred to as the "embedding" tree because it contains and summarizes the positions of samples in the lower-dimensional embedding. This tree is shown in orange in Fig. 3.7. A Linear-BVH is used here because it can be built quickly from a large number of points, and it provides tight-fitting bounding boxes for each node. The structure and construction algorithm for Linear-BVH is detailed in Section 3.3.2.

The second tree is referred to as the "field" tree because after traversal is completed, its nodes will contain a vector field representing the repulsive forces between points in the embedding. This tree is shown in blue in Fig. 3.7. This is a Quadtree or an Octree, tiling the same space as the embedding tree. Unlike the embedding tree, its nodes do not enumerate the points they contain. The quadtree is much shallower than the Linear-BVH and built to a fixed depth, so it can also be constructed quickly.

Using two hierarchies like this may appear to be an unnecessary cost, but it is advantageous because both trees are able to play to their strengths. The Linear-BVH

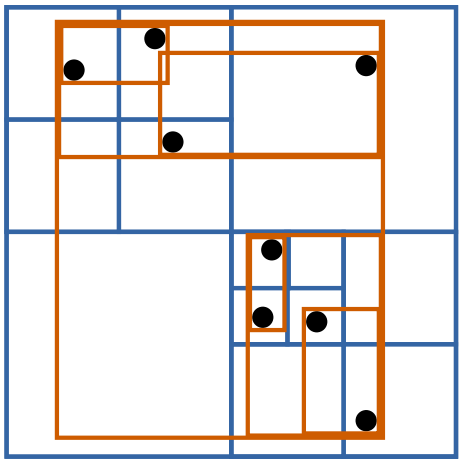


Figure 3.7: In dual hierarchy, the Linear-BVH active-tree (orange) and the Octree passive-tree (blue) partition the same space.

has tight-fitting nodes which more accurately indicate how dispersed the contained points are, allowing for more approximations than may otherwise be possible. The shallow Quadtree contiguously tiles the space and is always square, so the evaluation point when computing fields can be equidistant from all sides of the node. A Linear-BVH could not be substituted for the quadtree because its rectangular nodes make it more difficult to control error, and a quadtree could not be substituted for the Linear-BVH because building it to full depth would be more expensive.

To compute the acceleration values of the field tree, we perform a dual traversal, similar to that used in FMM, though the active and passive nodes do not come from the same tree in this case. An interaction between the two trees (drawn separately) is shown in Fig. 3.8. Note that the summary for the embedding node is always a "center of mass", which for t-SNE means an unweighted average of the positions of the contained points. The summary for the field tree is the center of the node.

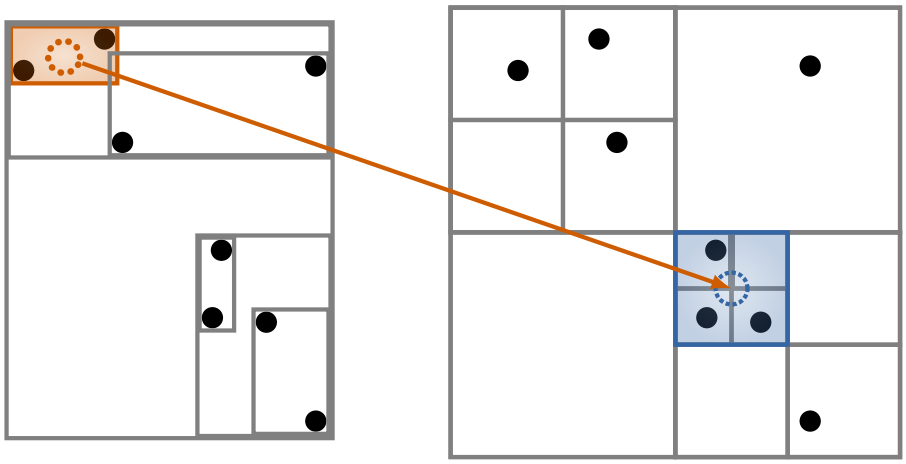


Figure 3.8: In dual hierarchy, long-range interactions are performed between nodes of the active tree (left) and those of the passive tree (right).

One approach for implementing this algorithm is shown in Algorithm 5. The biggest difference from the dual-traversal used in FMM is that the particles contained by the passive node are never handled individually, and instead the nodes collect all forces, even those at very close ranges. This means that the field is effectively rasterized as it is in Pezzotti et al. [2020] or Linderman et al. [2019].

Algorithm 5 A simplified implementation of t-SNE Dual Hierarchy traversal

```
1  procedure SEQUENTIAL_DUAL_TRAVERSAL(root_active, root_passive)
2      queue = [(root_active, root_passive)]
3      while SIZE(queue) > 0 do
4          (node_active, node_passive) = POP(queue)
5          if DESCENT_CRITERION(node_active, node_passive) then
6              REPULSIVE_FORCE(node_active, node_passive)
7          else
8              for each child_active ∈ node_active do
9                  for each child_passive ∈ node_passive do
10                     if IS_LEAF(child_passive) ∨ IS_LEAF(child_active) then
11                         for each [p_active] ∈ child_active do
12                             REPULSIVE_FORCE(p_active, child_passive)
13                     else
14                         PUSH(queue, child_active, child_passive)
```

The algorithm processes a queue of node pairs, each of which represents a potential interaction. It considers performing a direct interaction, and if that is not an option it descends the tree instead, adding each pair of children to the queue. When a leaf node is reached, it computes the exact forces on the passive (octree) node by interacting it with the particles contained by the active (Linear-BVH) node.

This version of the algorithm is somewhat simplified. The actual implementation runs entirely on the GPU, so traversal is split into several shaders which handle different parts of traversal and interaction. There is also additional handling for cases where leaves are reached, allowing the algorithm to decay to Barnes-Hut, in the same way as Algorithm 4 does.

3.3.1 The Projected Diagonal Descent Criterion

Another innovation of the Dual Hierarchy algorithm for t-SNE is its use of a *Projected Diagonal* descent criterion. The original Barnes-Hut algorithm and later FMM both use the side lengths of the nodes to determine whether an approximation can be accurate, but there are many ways this could be translated to the rectangular bounding boxes of a Linear-BVH. The length of the diagonal and the length of the longest side are both natural options.

The paper van de Ruit et al. [2022] discovers that t-SNE minimization convergence is more sensitive to errors of distance than to errors of direction. It is possible to exploit this and increase some forms of error while still producing a high-quality embedding in the same number of iterations. Instead of using the length of the diagonal directly, it is first projected onto the vector which connects the field sampling location to the active node, as shown in Fig. 3.9. This requires manipulating the signs of the connecting vector to ensure it is never perpendicular to the diagonal but is otherwise a straightforward operation.

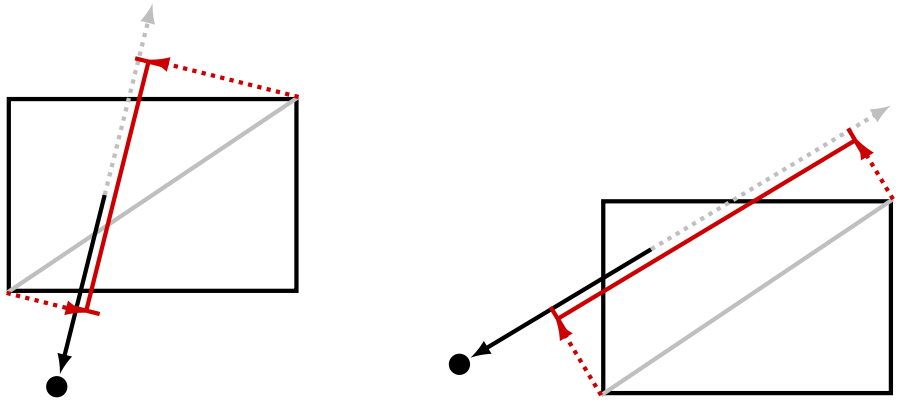


Figure 3.9: A pair of projected diagonals (red) for interactions with a rectangular node. Interactions along the longer side of the node have longer projected diagonals.

The resulting projected diagonal is short for connecting vectors which cross the node the shorter way, and longer for vectors that cross it lengthwise. The result is that the approximation criterion is significantly less strict for nodes that are "shallow" from the perspective of the point which receives the force, and more strict for nodes that are "deep".

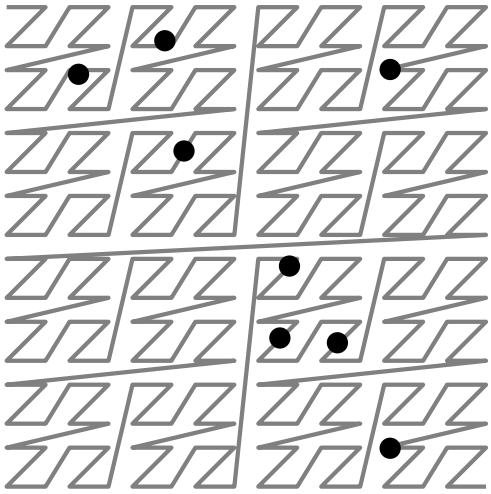
3.3.2 Linear BVH Construction

A Linear Bounding-Volume-Hierarchy (Linear-BVH) is a bounding volume hierarchy that can be constructed in linear time, using a technique first proposed by Arroyuelo et al. [2010]. The process is as follows:

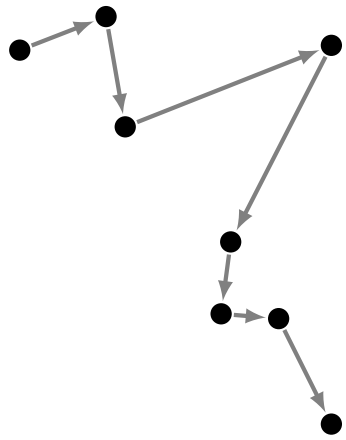
1) Each particle is assigned a Morton code. This is a number that uniquely places it along the Morton curve, pictured in Fig. 3.10a. The Morton curve (also known as a "Z-curve") is a space-filling curve like the Hilbert curve, with the added benefit that Morton codes can be generated simply by interleaving the bits of the x, y, z, components of a Cartesian coordinate. As a result, this can be done in linear time using only a handful of bitwise operations.

2) Particles are sorted by their Morton codes using a Radix sort. This step has a complexity of $O(nd)$, where d is the number of digits in the Morton code. Dual Hierarchy for t-SNE always uses 32-bit codes, so this is also linear time. Notice how in Fig. 3.10b, the Radix sort of Morton codes orders the particles such that particles which are nearby in the high-dimensional space are also close in the sequence.

3) We build the bounding volume hierarchy from the bottom up. We first produce a node from each pair of adjacent particles in our sequence, and then from each pair of adjacent nodes, and so on. For the leaf nodes, we produce bounding boxes that tightly fit the contained particles (Fig. 3.11a). This must account for every particle and therefore has linear complexity. For internal nodes, we can produce bounding boxes that tightly fit the bounding boxes of the children (Fig. 3.11b); doing this for each level of the tree has complexity $\log(n)$. The overall complexity of this step is therefore also linear.

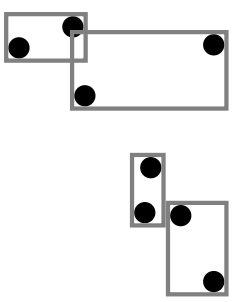


(a) Each particle can be assigned a Morton code based on its position.

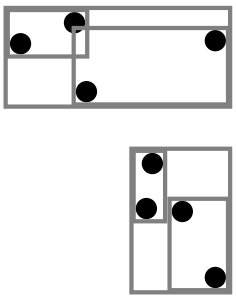


(b) Sorting by Morton codes places nearby particles nearby in the resulting sequence.

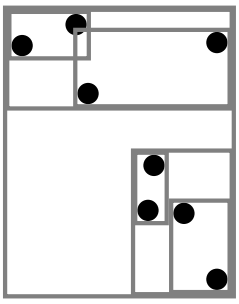
Figure 3.10: Morton-sort for locality-preserving dimension reduction.



(a) Leaf nodes contain adjacent particles.



(b) Interior nodes contain adjacent child nodes.

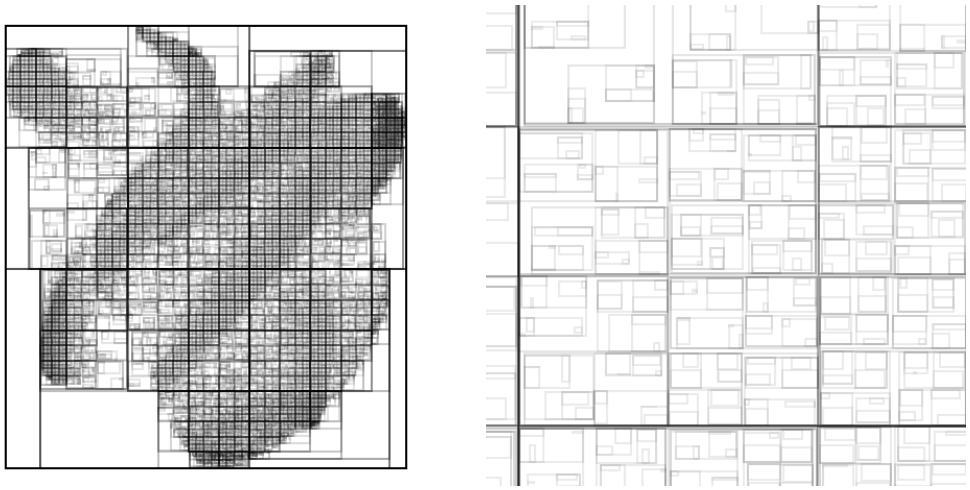


(c) Root node contains the entire sequence of particles.

Figure 3.11: Bottom-up construction of a linear-BVH from a sorted list

For a higher-quality tree, we partition the particles from the top down. At each level, we split based on the greatest different bit in the Morton codes. This requires a binary search for each node on a level of the tree ($\log(n)$) for each level of the tree ($\log(n)$), so it is also sub-linear. This avoids the large jumps in the Morton curve, producing nodes with lower aspect-ratio bounding boxes and a structure more similar to that of an Octree. A tree produced using this method in 2 dimensions is shown in Fig. 3.12. Notice how non-leaf nodes are closer to square; higher aspect ratios are only produced as a result of tightly fitting a small number of particles.

An Linear-BVH has an additional advantage over other trees because the bottom-up portion of construction can be run separately from the sorting step. This means that if the particles the tree is built from only move by a small amount, the bounding boxes can be updated to produce a valid tree in less time. This is naturally a lower-quality tree, but when the change in position is small enough the difference in tree quality can be outweighed by the reduced construction time.



(a) The complete tree, for 70,000 samples (b) A zoomed-in view of part of the tree

Figure 3.12: A 2D Linear-BVH, constructed over the points of a t-SNE embedding during minimization.

3.4 Accuracy Metrics for Gravitational *n*-body

An appropriate measure of accuracy is critical in order to fairly evaluate the dual-tree algorithm against other approaches. Because accuracy can be exchanged for performance by changing the hyperparameter θ, even a very inefficient tree algorithm could be made to outperform FMM by disregarding accuracy and setting θ very high.

An intuitive method for measuring the accuracy of an *n*-body solver is to simulate a number of time steps with both the naive algorithm and the candidate solver, and then compare the positions of each particle. Unfortunately, this approach is not viable in practice. The collection of point masses simulated in the *n*-body problem is a chaotic system (Boccaletti and Pucacco [1998]). As a consequence, even very small differences such as those produced by rounding errors compound over multiple steps. This limits the ability of even the naive *n*-body solver to make predictions far into the future, but it also makes it impossible to separate out the errors introduced by different solving methods.

To avoid this problem, the generally accepted approach is to perform static analysis. By measuring the accuracy of the force applied to each particle, we can determine the accuracy of an individual time step.

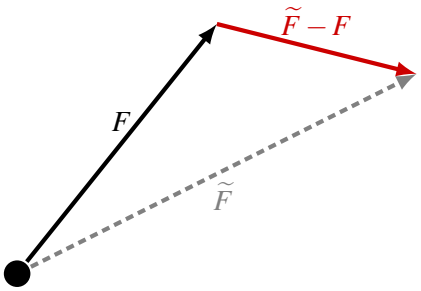


Figure 3.13: Exact and approximate force vectors, with the error vector in red.

The force error is closely related to the acceleration error, but it has the additional advantage of also taking the mass of each particle into account. In effect, each error is weighted by mass, so that heavier particles are more sensitive to error. Most n-body solver implementations compute acceleration directly rather than computing force first and then dividing by mass, but forces are trivially derived from mass & acceleration. This approach is shown in Fig. 3.13, where F is the true force, and $\widetilde{F}$ is the approximate force.

We experimented with several measures of error, used in different survey papers. We discovered that measures of force error are strongly correlated. Definitions of acceptable error may be very different for different measures, but changes to an algorithm which increase one error measure will typically also increase the others, and vice versa. For the purpose of benchmarking, we selected a pair of the most useful error metrics. In Section 3.4.1 we discuss RMS Error, and its advantages and limitations. In Section 3.4.2 we discuss the "Constitutional error" metric, its advantages, and the properties which make it especially strict compared to other metrics.

3.4.1 RMS Error

RMS error is based on the root of the sum of all force errors squared (shown in Fig. 3.14). In practice, this is usually expressed as a percentage. The RMS of the force errors is compared to the RMS of the force magnitudes. In other words, the goal is to limit the L2 norm of the magnitudes of the force errors to a set percent of the L2 norm of the magnitudes of the forces.

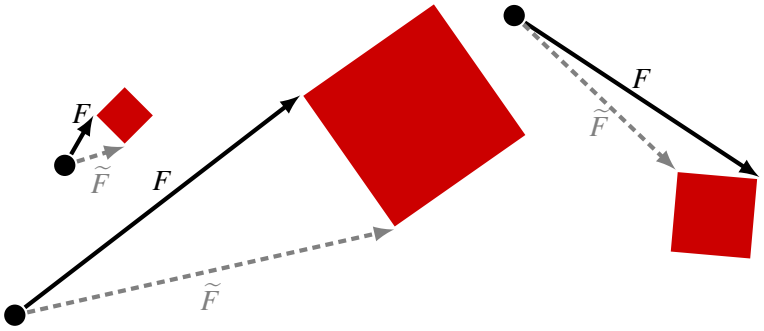


Figure 3.14: Exact and approximate force vectors, with the squared error for each in red. The RMS error is based on the sum of the area of all squares.

RMS error is the metric of choice for many algorithm designers, but the actual threshold percent must be chosen based on the specific application. For example, PD-KGRAV3 is designed for an especially strict limit of 0.1%, while Springel et al. [2021] mentions a target of sub-1%. Agreeing on a threshold is difficult because not all error is equivalent – correlated or otherwise structured error can cause greater problems. Moreover, it is impossible to compare most implementations against the naive algorithm at the scales for which they were designed. The Schneider et al. [2016] survey attempts to resolve this by comparing the implementations against one another instead, and also by measuring overall power spectra error instead of individual force errors. For large-scale simulations, it sees discrepancies of greater than 1% between several leading algorithms.

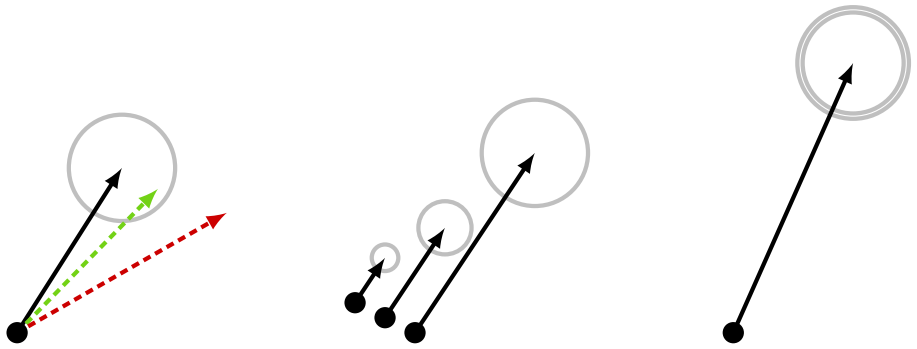
3.4.2 Constitutional Error

Constitutional error is defined in an appendix of Lake et al. [1995], titled "The *n*-body Constitution", and used in the survey Dikaiakos and Stadel [1996]. The goal of the document is to define a set of conditions that, if met, guarantee that an approximation approach will produce accurate simulations of real data. The document includes a variety of requirements, ranging from the time step size, to the integrator, to properties of the simulated dataset. The most relevant requirement for the purposes of this report is described in Article II: "The error should always be less than 0.5% of the force or the RMS force, whichever is less". This condition is written in more formal terms in Eq. (3.7).

$$\|F - \widetilde{F}\| < 0.005 \cdot \text{MIN}(\|F\|, \text{RMS}) \tag{3.7}$$

Where F is the true force on a given particle, calculated using the naive method, $\widetilde{F}$ is the approximate force on that particle, calculated using the candidate solver, and RMS is the root of mean squares of the magnitudes of all true forces in the simulation.

This criterion has several properties which make it preferable over metrics such as RMS Error. The consequences of implementing the criterion in this way can be seen in Fig. 3.15.



(a) Each force has a radius of allowable error.

(b) Allowable error is proportional to magnitude.

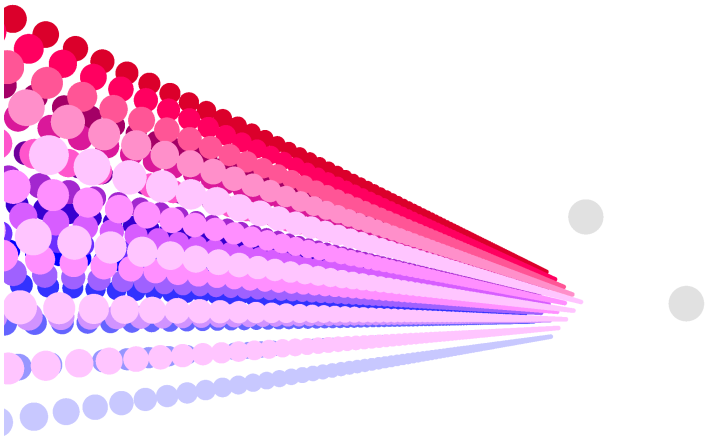
(c) For large forces, allowable error is capped.

Figure 3.15: A simple visualization of the Constitutional Error metric for 2-dimensional force vectors.

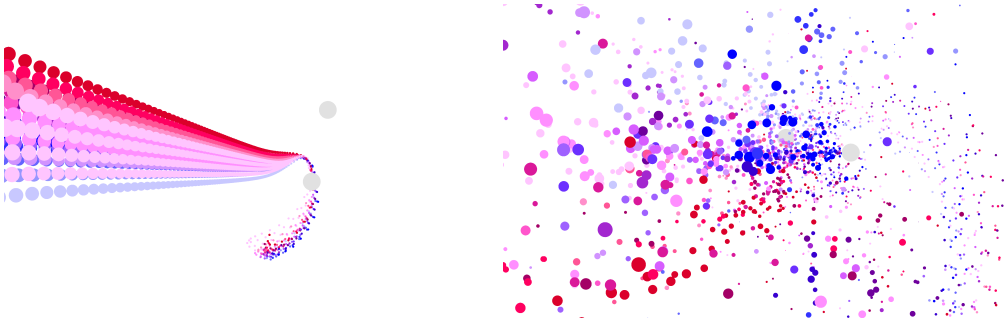
3.4.3 Dataset Selection

The scenario used for testing can have significant effects on performance and accuracy. As shown in Table 4.1, the distribution of point masses can drastically affect the structure of the hierarchies used by the algorithms being tested, and this changes which approximations are possible. Properties of the distribution also affect how accurate those approximations are on average.

For initial functionality tests, we used a collection of small hand-written scenarios. These range from small systems of orbiting particles to larger scenarios such as the one shown in Fig. 3.16. Cubic grids of particles are produced programmatically, and colored to make it easy to track the movement of particles by eye. All are small enough



(a) Initial state; cubic grid of 2,500 small particles, with binary system of heavy "sun" particles.



(b) After 500 iterations.

(c) After 5,000 iterations.

Figure 3.16: Snapshorts of a hand-written dataset at different stages of simulation

to run quickly using the naive approach, and the regular layout makes it easy to catch divergence from correct results within a small number of iterations. This was useful for debugging while adapting the algorithm; for example, if certain forces are double-counted or ignored by the treecode, this is immediately obvious.

For small-scale correctness tests, we use a uniform randomly generated distribution, as shown in Fig. 3.17. To produce each particle, X, Y, and Z coordinates are independently selected from a uniform distribution. Because the error is calculated proportionally and our gravity equation is unitless, the scale doesn't matter. Ranges are selected for a given value of n such that the average density is equivalent to 1 particle per u^3. This is necessary for iterative error metrics because "crowded" datasets require very small time step sizes to maintain accuracy, but does not make a difference for the static analysis approach which was ultimately used. Velocities are also chosen from a uniform distribution and also matter only for iterative tests. The mass of each particle was selected from an exponential distribution with a mean of 1.0, so that the scenario is primarily composed of low-mass particles, with a small number of much heavier particles.

The uniform distribution is useful for some types of testing, but it can be misleading when evaluating tree performance. The even distribution produces unrealistically well-balanced octrees and reduces the likelihood of particles with the same Morton code when building a Linear-BVH. To resolve this, for many small-scale tests we use

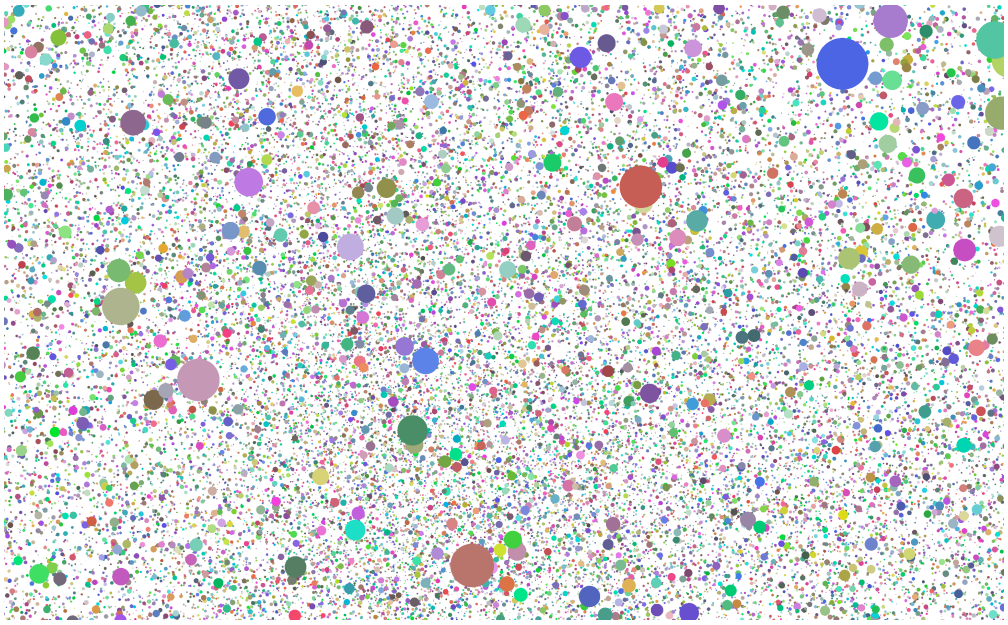


Figure 3.17: A random dataset with uniform distribution, 100,000 particles with radii based on mass and colors chosen at random.

randomized datasets based on Perlin noise, as shown in Fig. 3.18. Generation is done in much the same way as for the uniform distribution, but we sample Perlin noise at each position. If the 3D Perlin function is below a set threshold for that position, we reject the particle and do not add it. We sample a coarse Perlin function to produce large-scale noise and add a fine-grained function to prevent hard boundaries between high-density and empty regions. Because many particles are rejected, Perlin noise datasets take longer to generate.

Perlin noise datasets are useful for testing solvers on different tree structures but still do not provide a realistic analog for actual cosmological simulations. Galaxy and galactic neighborhood simulations can have very high "dynamic range", meaning that there are both areas of extremely high density and areas that are nearly empty. This is doubly true for so-called "extreme zoom" simulations, which contain structures at a variety of scales. TreePM handles extreme dynamic range well, but in order to design a treecode that can be integrated into a Particle-Mesh based solver, we need to ensure it can perform well on scenarios with moderately high dynamic range.

For this purpose we use the AGORA dataset from Kim et al. [2016] when performing middle- and large-scale tests. The AGORA dataset comes in three sizes: AGORA-LOW contains 112,500 stars, MED contains 1,125,000 stars, and HI contains 11,250,000 stars. The smallest AGORA dataset is shown in Fig. 3.19. Each particle has the same mass and represents a cohort of stars.

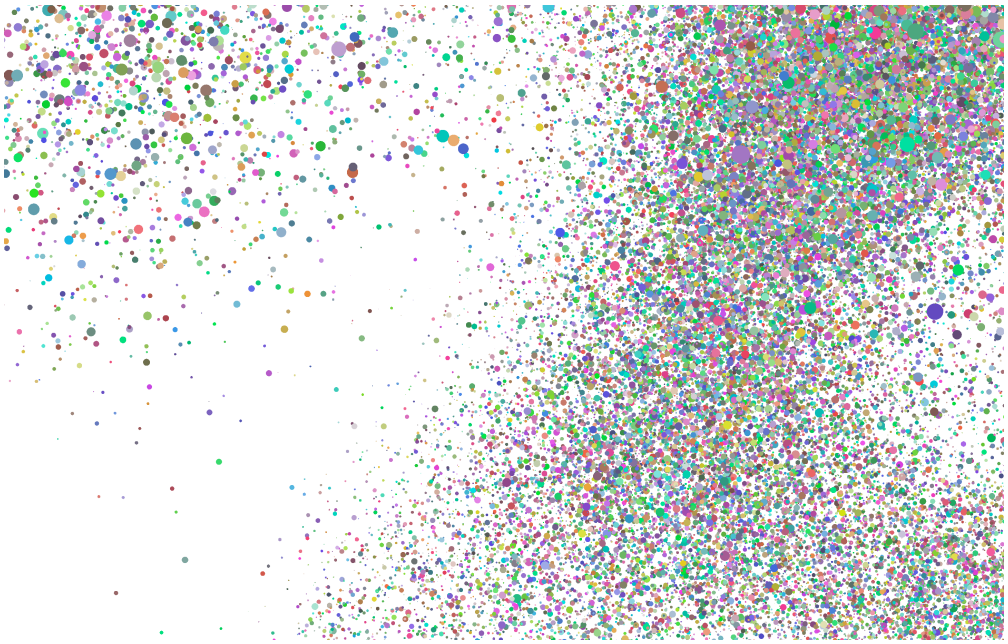


Figure 3.18: Perlin noise dataset, 50,000 particles with randomized radius and color, positions selected randomly, existence determined by a threshold on Perlin noise.

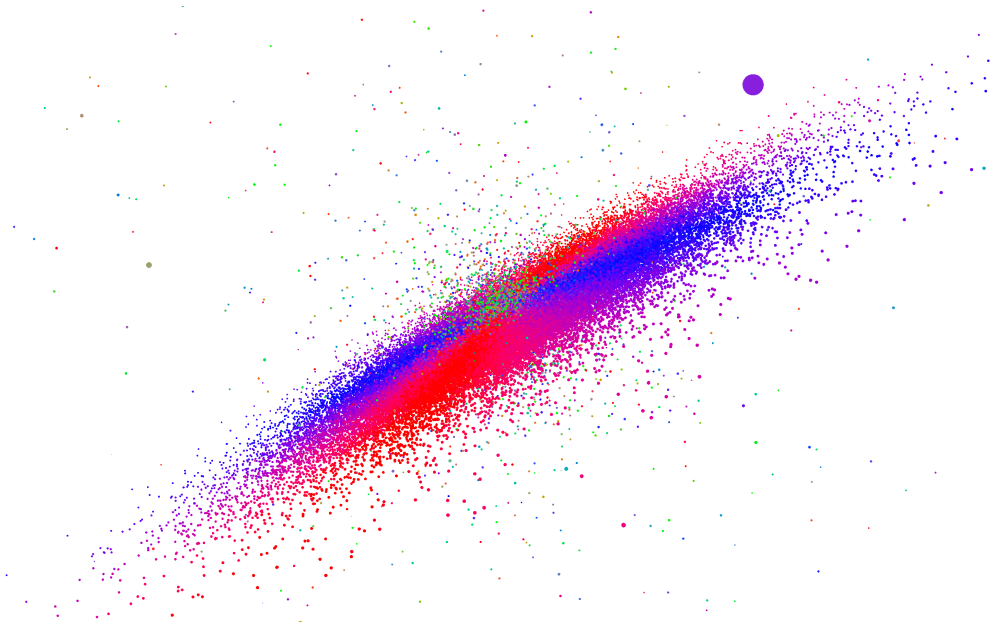


Figure 3.19: AGORA dataset, 112,500 particles with uniform radius and color indicating velocity and direction.

3.5 Multipoles

Contemporary gravitational *n*-body solvers improve accuracy by using multipole approximations in place of center of mass and uniform acceleration fields.

A multipole is a sequence of symmetric tensors of ascending rank. A multipole of order 1 in $\mathbb{R}^3$ space contains only an $\langle X, Y, Z \rangle$ vector, a multipole of order 2 contains a vector and a 3×3 symmetric matrix, and higher order multipoles contain additional symmetric tensors of progressively higher rank.

Here, symmetric is defined such that in a rank-2 tensor, $q_{xy} = q_{yx}$. This means that a quadrupole symmetric tensor can be written without redundancy by only including the upper triangle, as shown in Eq. (3.8).

$$\begin{bmatrix} q_{xx} & q_{xy} & q_{xz} \\ \cdot & q_{yy} & q_{yz} \\ \cdot & \cdot & q_{zz} \end{bmatrix} \tag{3.8}$$

As a result, a dense implementation only needs to store 6 values instead of nine. This becomes progressively more important with higher-order multipoles. An octupole ($3 \times 3 \times 3$ symmetric tensor) can be stored with 10 values instead of 27 by only including one pyramid, as shown in Eq. (3.9). A hexadecupole can be stored with 15 values instead of 81 by only including one hyper-pyramid.

$$\left[\begin{bmatrix} o_{xxx} & o_{xxy} & o_{xxz} \\ \cdot & o_{xyy} & o_{xyz} \\ \cdot & \cdot & o_{xzz} \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & o_{yyy} & o_{yyz} \\ \cdot & \cdot & o_{yzz} \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & o_{zzz} \end{bmatrix} \right] \tag{3.9}$$

For some multipole gravity implementations such as Rein and Liu [2012], the symmetric tensors are also guaranteed to be traceless. Given that $q_{xx} + q_{yy} + q_{zz} = 0$, it is possible to omit q_{zz} from the definition and derive its value from q_{xx} and q_{yy} only when necessary. In practice, the memory advantage is marginal and unlikely to outweigh the computation cost of finding q_{zz} each time it is needed.

Field Plots

In order to understand the use of multipoles in practical terms, it is useful to look directly at the gravitational fields they attempt to approximate. For simplicity, we only look at a 2-dimensional "slice" of the gravitational field, and the in-plane component of the acceleration. A traditional vector field would be an appropriate approach here, but we can produce a much more detailed plot if we use color to indicate direction, as shown in Fig. 3.20.

This technique is demonstrated in Fig. 3.21, which shows the the x and y components of the gravitational field at $z = 0$ across a small simulation domain. Importantly, the zoomed region in Fig. 3.21b is re-scaled so that the strongest acceleration *within the region* is assigned full saturation.

3.5.1 Multipole Moments

A multipole moment is a way of summarizing a group of point masses more accurately than their center of mass. It represents the moment of inertia of the cluster along a

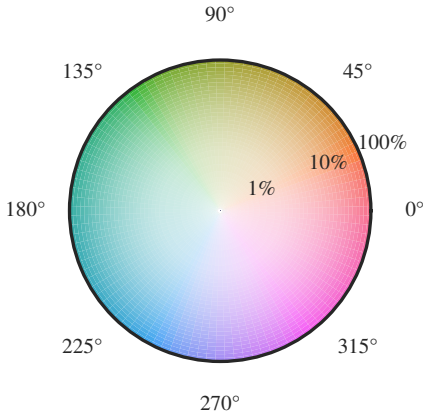
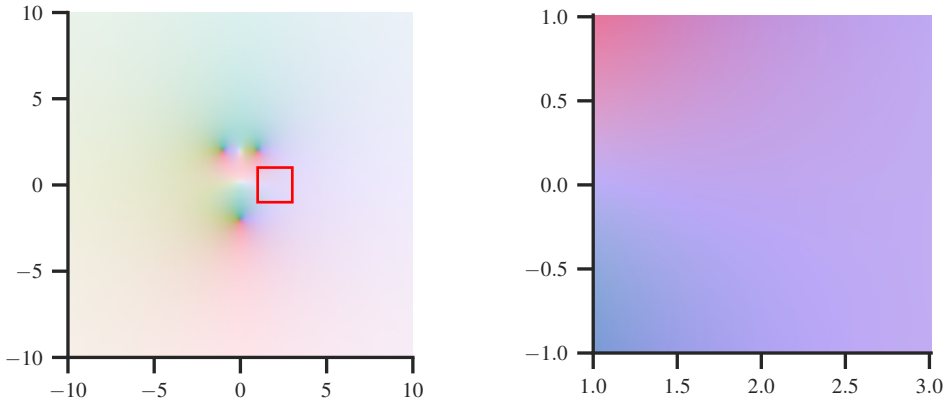


Figure 3.20: Key for interpreting field plots. Color corresponds with the radial direction of the acceleration of gravity, and saturation indicates its magnitude, expressed as a percentage of the maximum magnitude shown in the plot.



(a) The field surrounding the trio of particles (b) A zoomed view of the area outlined in red.

Figure 3.21: The exact gravitational field of a trio of particles, rendered using the method defined by Fig. 3.20.

variety of axes. Simply put, it encodes information about how spread-out the particles are in different directions. The center of mass is still included in calculations, but the moment provides additional information about how the point masses are distributed in relation to the center of mass.

For multipole moments, we include a scalar net-mass term in addition to the series of tensors, so a rank-2 (quadrupole) moment has the format shown in Eq. (3.10).

$$m,\ \langle m_x, m_y, m_z \rangle,\ \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} \\ \cdot & m_{yy} & m_{yz} \\ \cdot & \cdot & m_{zz} \end{bmatrix} \tag{3.10}$$

To find the multipole moment representation for a group of particles, we begin by computing the total mass and center of mass, just as we would for a simple center of mass summary. We can then find the moment of inertia for each contained point mass individually. The individual moments are given by the cartesian power of the offset from the center of mass $\Delta = \langle d_x, d_y, d_z \rangle$, equivalent to the repeated outer product of Δ with itself. This tensor is weighted by the mass of the particle. The quadrupole moment equation is shown as an example in Eq. (3.11)

$$Q = m\Delta^2 = m \begin{bmatrix} d_x d_x & d_x d_y & d_x d_z \\ \cdot & d_y d_y & d_y d_z \\ \cdot & \cdot & d_z d_z \end{bmatrix} \tag{3.11}$$

The net moment of the cluster is given by the simple element-wise sum of the multipole moments of individual particles.

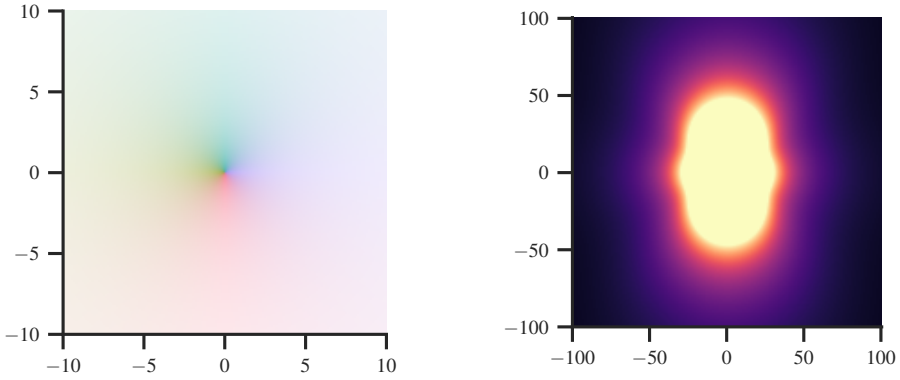
Generating multipole moments for non-leaf particles can be done faster by summarizing child nodes instead of every contained particle. Given that this is done recursively, we already have the multipole moments of each child node. We can combine these moments in much the same way the moments of individual particles were combined, but first the moments must be in relation to a common center of mass. To do this, we find the moment of the entire child node using the offset of its center of mass in relation to the center of mass of the parent. This moment is added to the moment of the child, applying the appropriate offset to its center of mass and to each of its symmetric tensors. This combined multipole can then be added to the net moment of the parent node.

In Practice

To better understand what multipole moments accomplish, we can use a series of multipoles to represent the trio of particles shown in Fig. 3.21a. To show the benefits of higher-order multipoles, we also need to show their effect on error. For this, we supplement the field plots with corresponding error plots, using the scale shown in Fig. 3.22.

To motivate the usefulness of multipole moments, we can first look at the performance of a center-of-mass representation. The field produced by such an approximation is shown in Fig. 3.23a, and is equivalent to the field produced by a single particle with its mass the sum of the masses it approximates. Notice how the approximation only becomes acceptably accurate at very long ranges, and it is necessary to use a larger scale for Fig. 3.23b. This corresponds to a need for very low values of θ.

Figure 3.22: Key for interpreting field error plots. Error is given as a percent of the true acceleration, and displayed values are always capped at 0.5%, as any greater error is guaranteed to violate the constitutional criterion.



(a) Center-of-mass representation

(b) Center-of-mass relative error

Figure 3.23: Center-of-mass approximation of the field shown in Fig. 3.21a, alongside the error of this approximation across a much larger region.

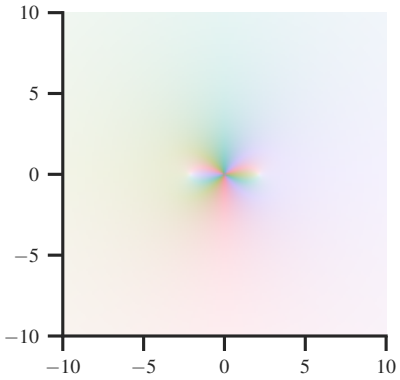


Figure 3.24: Quadrupole moment approximation of the same field.

We can produce a more accurate representation of our trio of particles by adding a Quadrupole term. The resulting field is shown in Fig. 3.24. Notice how the higher order term dominates for close-range forces, and the field is more complicated close to the center, where the two field components produce additive and subtractive interference. We only use the approximation for longer-range forces, in regions of the field where the quadrupole component is much smaller than the center-of-mass component.

In the left column of Fig. 3.25, we show each multipole component independently; this makes it easier to see the way in which they affect the field. Each component makes progressively finer adjustments, but because the center-of-mass approximation is near-accurate across a broad range, even a small change can significantly shrink the radius at which it becomes inaccurate. The exact shape of the field error shown in the right column is unique to the arrangement of particles being represented. The region of high error shrinks with each higher-order multipole, and this effect generalizes to other arrangements.

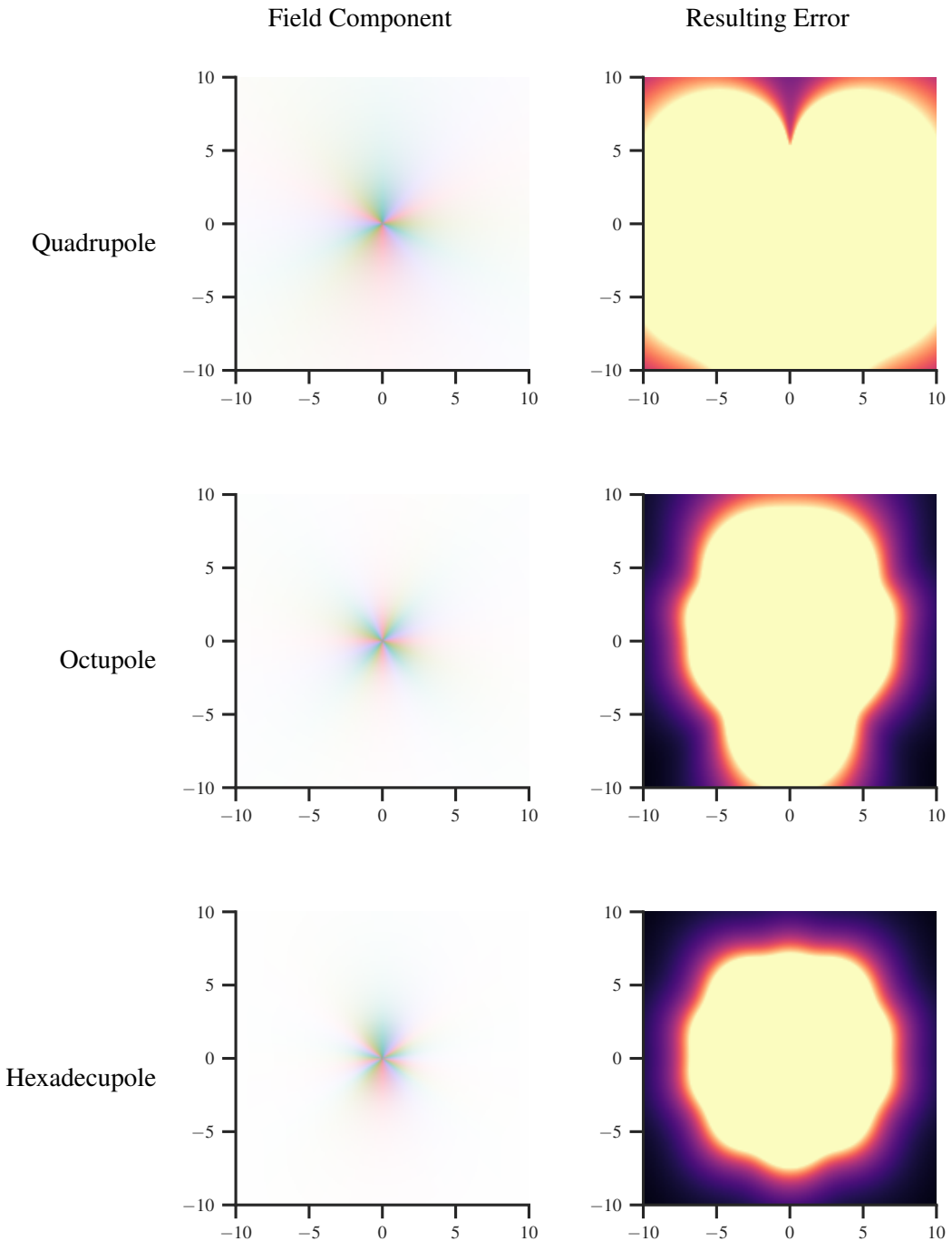


Figure 3.25: Field and error plots for multipole moments of several orders.

3.5.2 Field Multipoles

Multipole moments alone are enough to massively improve the accuracy of single-tree algorithms like Barnes-Hut, but they are not sufficient to implement true FMM. For that, we also need to use a multipole representation for acceleration.

A multipole acceleration is a series of derivatives of the acceleration of gravity, with respect to changes in sampling location. For example, the quadrupole component represents the first derivative. Because we are taking the derivative of a vector with respect to another vector, the result is a matrix:

$$\langle a_x, a_y, a_z \rangle, \quad \begin{bmatrix} \frac{da_x}{dx} & \frac{da_x}{dy} & \frac{da_x}{dz} \\ \cdot & \frac{da_y}{dy} & \frac{da_y}{dz} \\ \cdot & \cdot & \frac{da_z}{dz} \end{bmatrix} \tag{3.12}$$

Here, the term $\frac{da_x}{dy}$ is the slope of the linear change in a_x with respect to change in y. This is equivalent to the inverse $\frac{da_y}{dx}$, so we can still use a symmetric matrix. Each time we take the derivative with respect to x, y, and z another dimension is added, so higher-order multipoles are necessary to represent progressively higher order. To find values for the quadrupole field, we need to use an expression for the derivative of gravity, dependent on the connecting vector R, this is written as $D_2(R)$. This expression and those of higher-order derivatives can be found in Springel et al. [2021].

When applying a multipole acceleration to each of the particles it applies to, it must be re-centered from the sample position to the location of the particle. We assume the derivatives to be uniform across the node, so we only need to update the vector component $\langle a_x, a_y, a_z \rangle$. For a quadrupole field, this is as simple as adding our matrix contracted with the shift in sample position $\Delta = \langle d_x, d_y, d_z \rangle$:

$$\langle a_x, a_y, a_z \rangle \;=\; \langle a_x, a_y, a_z \rangle \;+\; \begin{bmatrix} \frac{da_x}{dx} & \frac{da_x}{dy} & \frac{da_x}{dz} \\ \cdot & \frac{da_y}{dy} & \frac{da_y}{dz} \\ \cdot & \cdot & \frac{da_z}{dz} \end{bmatrix} \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \tag{3.13}$$

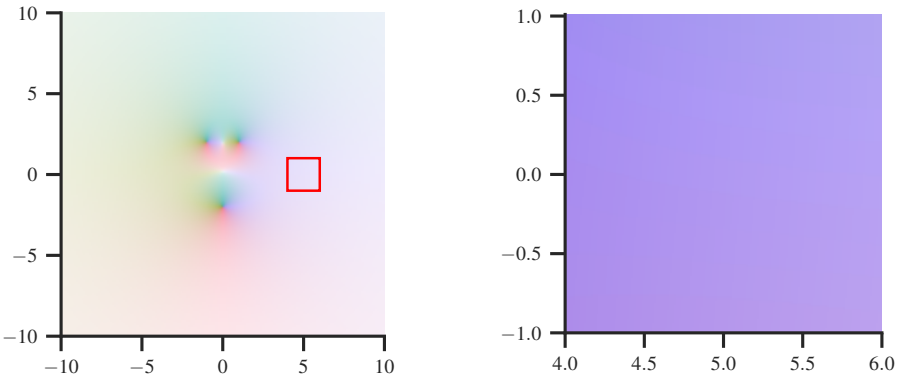
For higher-order components, we perform repeated contractions and need to include Taylor coefficients, but the process is generally similar.

The acceleration tree contains multipoles which are one order higher than the force tree. This is because even a single point-mass (which can be represented by its center-of-mass with zero error), produces a non-uniform gravitational field, which can't be accurately represented by a single acceleration vector.

In Practice

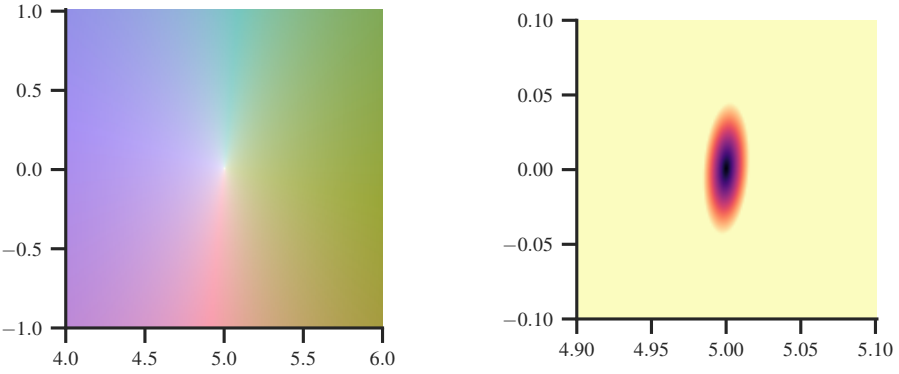
To understand how multipole fields are used to improve accuracy, we can look at multipole representations of the field within the domain shown in Fig. 3.26b. In many ways, field approximations are best understood as the inverse of mass approximations. While mass approximations have zero error at infinite distance, field approximations have zero error at the sampling location. Our goal for mass approximations is to shrink the region of high error; our goal for field approximations is to do the opposite, growing the region of low error.

For mass approximations, a center-of-mass (vector) approach works, albeit poorly, as shown in Fig. 3.23. A vector approximation of the field implies a uniform acceleration across the node, and this performs much worse. While Fig. 3.26b may appear nearly uniform, the subtle gradient makes a significant difference. Even a single mass produces a non-uniform field, so error grows rapidly with distance from the sampling location, and even infinitesimal distances cause us to fail the constitutional error condition.



(a) The field surrounding the trio of particles (b) A zoomed view of the area outlined in red.

Figure 3.26: The exact gravitational field of a trio of particles, with a selected region to be approximated using Field Multipoles.



(a) Quadrupole component (b) Quadrupole relative error

Figure 3.27: Isolated quadrupole component of an approximation of the field shown in Fig. 3.26b, alongside the error of this approximation across a much smaller region.

We can improve the situation somewhat by using a quadrupole, as shown in Fig. 3.27. The quadrupole component is shown in Fig. 3.27a, and its appearance matches intuitions for a representation of the first derivative of acceleration. At the sampling location in the center of the image, the uniform approximation is accurate, so the

quadrupole makes no change. The region to the left (closer to the trio of particles) agrees with the uniform approximation, resulting in a stronger field. The region to the right does the opposite. Unfortunately, this first-order representation still provides poor accuracy. In Fig. 3.27b, the approximate field becomes unacceptably inaccurate at small distances from the sampling location, and it is necessary to use a smaller scale to see the region of acceptable accuracy.

Figure 3.28 shows progressively higher order components of multipole fields in the left column and the resulting error across the same domain in the right column. It is worth noting that the error plots show a region $10\times$ larger than that shown in Fig. 3.27b, meaning that even the Octupole representation is significantly more effective than the Quadrupole.

3.5.3 Moment-Field Interactions

Dual-tree and dual-traversal algorithms involve node-node interactions. That means that to implement the Fast Multipole Method or the dual hierarchy algorithm we need to compute the interactions between multipole moment and field representations.

From Springel et al. [2021], the quadrupole version of this interaction is given by Eq. (3.14)

$$Q_{field} \simeq -G\left[mF_g + \frac{1}{2}Q_{moment}\cdot F_g^{(2)} - mF_g^{(1)}\cdot\Delta + \frac{1}{2}Q_{moment}\cdot\Delta^2\right], \tag{3.14}$$

Here, Δ is the vector connecting the centers of mass of the two nodes, and $F_g^{(n)}$ is the n-th derivative of gravity, evaluated at the sample position. The complete list of these derivatives can be found in Appendix A.

Once this is implemented, we also have node-particle and particle-node interactions because these can be understood as special cases of the multipole-multipole interaction. For interactions between a multipole-moment and a particle, we only take the vector component of the field multipole. For interactions between a particle and a multipole field, we first expand the point mass into a multipole.

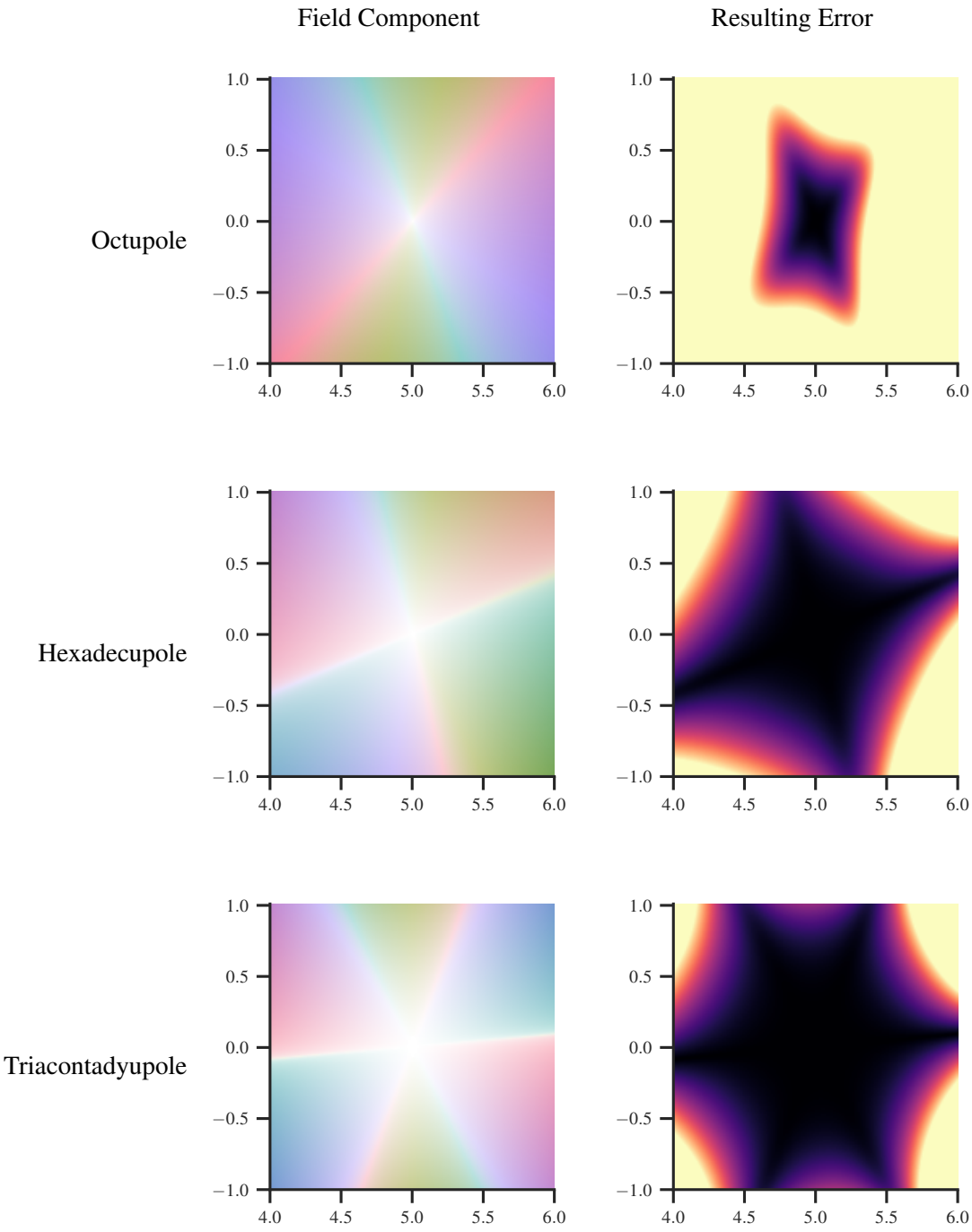


Figure 3.28: Field and error plots for multipole fields of several orders. Sampling location is at the center of each field

Chapter 4

Method

In this section, we explain how the Dual Hierarchy method was adapted for cosmological n-body simulations, and provide details of our particular implementation.

Section 4.1 describes a pair of unusual algorithms for gravitational n-body which, when combined, make up the Dual Hierarchy algorithm. Section 4.2 lists the major differences between the t-SNE and gravitational variants of n-body, and explains how each of these differences are addressed by features of the adapted algorithm. Section 4.3 goes into more detail on how the algorithm and its data structures are implemented in C++, and Section 4.4 describes several changes which improved the performance of the algorithm. Finally, the adapted algorithm has hyperparameters that affect performance and accuracy, and Section 4.5 explains how these were chosen for best results.

4.1 Intermediate Algorithms

Before implementing the Dual Hierarchy algorithm, it was useful to explore certain related algorithms. These algorithms have no use in practice and do not appear in any of the surveys discussed, but they are useful as tools for understanding the complete Dual Hierarchy method. Together, they form the two sides of the dual-traversal approach, and in order for the final algorithm to be fast and accurate, both of these component parts must also work correctly. Section 4.1.1 describes a variation on Barnes-Hut which is equivalent to the force-emitting side of the dual hierarchy algorithm, and Section 4.1.2 describes another variation which comprises the field-tree side.

4.1.1 Linear-BVH Barnes-Hut

Linear-BVH Barnes-Hut uses the same active-tree traversal shown in Algorithm 2, but the octree is replaced by a linear-BVH, as shown in Fig. 4.1.

Because the nodes of a Linear-BVH are not always cubic, the descent criterion must be replaced with one which depends on either the longest side length of the node or the length of its diagonal. Potential descent criteria are explored in detail in Section 4.2.3.

Implementing Linear-BVH Barnes-Hut validates the active side of the Dual Hierarchy algorithm. The final algorithm also does an active-tree descent over a Linear-BVH,

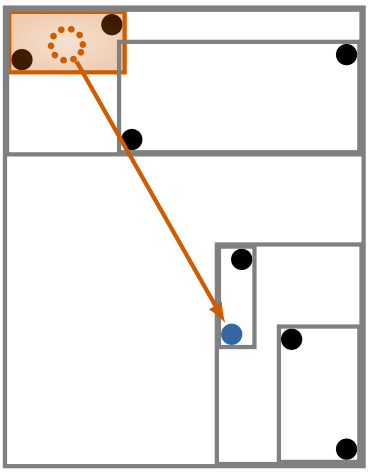


Figure 4.1: In Linear-BVH Barnes-Hut, long range forces use the center-of-mass of nodes in a Linear-BVH.

and in order for that to be accurate it must also produce correct results when used independently. The final recursive Dual Hierarchy implementation actually invokes the same traversal function which implements the Linear-BVH Barnes-Hut method when a leaf of the passive tree is reached.

4.1.2 Reverse Barnes-Hut

Reverse Barnes-Hut uses a passive "field" tree containing accelerations, rather than an active tree containing masses. Where Barnes-Hut iterates over each particle, and collects the forces on that particle from the tree, Reverse Barnes-Hut iterates over each node and collects the forces on that node from the list of particles.

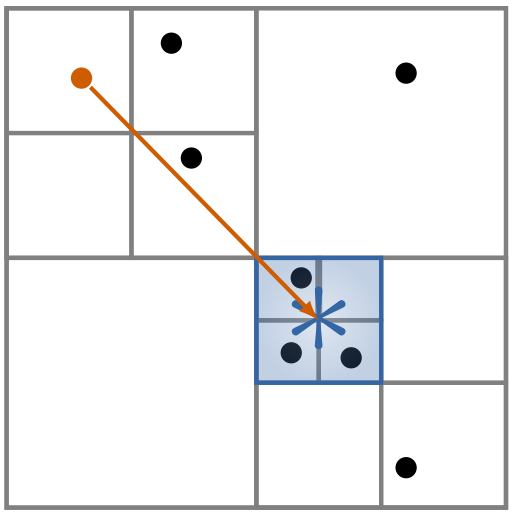


Figure 4.2: In Reverse Barnes-Hut, passive nodes (blue) receive long-range forces from individual active particles (orange).

This requires an inversion of the descent logic; large nodes collect forces from faraway particles, and smaller nodes collect forces from closer particles. In Fig. 4.2, a

mid-sized node receives a force from a particle. As with Barnes-Hut, near-field forces decay to direct interactions. One potential implementation of this descent is shown in Algorithm 6.

Algorithm 6 A recursive implementation of Passive-Tree Traversal

```
1  procedure TRAVERSE_PASSIVE(p_active, m_active, node)
2      if DESCENT_CRITERION(p_active, node) then
3          [p_node, a_node] := node
4          a_node ← a_node + GRAVITY(p_active, m_active, p_node)
5      else if IS_LEAF(node) then
6          for each [p_passive, a_passive] ∈ node do
7              a_passive ← a_passive + GRAVITY(p_active, m_active, p_passive)
8      else
9          for each child ∈ node do
10             TRAVERSE_PASSIVE(p_active, m_active, child)
```

After the traversal defined in Algorithm 7 completes, the nodes of the passive tree will each contain an acceleration. This is the net ambient acceleration of gravity received by that node from particles far enough away that the approximation was possible. Large nodes shallow in the tree will hold the acceleration due to very long-range forces, and nodes deeper in the tree will hold the acceleration due to more local particles. Very close-range forces will be applied directly to the particles, and are not included in the tree.

Before we can update the positions and velocities of each particle, we need to "collapse" the field tree so that each particle receives the sum of accelerations from all nodes of which it is a member. To do this, we can use the recursive algorithm shown in Algorithm 7. Each node distributes its acceleration to all children, and when a leaf node is reached the total acceleration is applied to all member particles.

Algorithm 7 A recursive implementation of Passive-Tree Collapse

```
1  procedure COLLAPSE_PASSIVE(node, a_net)
2      [p_node, a_node] := node
3      a_net ← a_net + a_node
4      if IS_LEAF(node) then
5          for each [p_particle, a_particle] ∈ node do
6              a_local ← EVALUATED_AT_OFFSET(a_net, p_particle − p_node)
7              a_particle ← a_particle + a_local
8      else
9          for each child ∈ node do
10             [p_child, a_child] := child
11             a_local ← EVALUATED_AT_OFFSET(a_net, p_child − p_node)
12             COLLAPSE_PASSIVE(child, a_local)
```

If a_{node} and a_{net} are simple vectors, then EVALUATED_AT_OFFSET() does nothing. In practice, multipole fields are necessary for reverse Barnes-Hut to produce accurate results. When a_{node} and a_{net} are multipole fields, EVALUATED_AT_OFFSET() finds the appropriate re-centered multipole using Eq. (3.13).

We can build a reverse Barnes-Hut solver by applying these two steps to an octree as shown in Algorithm 8. Each force-emitting particle is incorporated into the ambient

gravitational field represented by the tree. We collapse the tree starting at the root, with an initial global acceleration that is typically zero.

Algorithm 8 Application of Passive-Tree traversal to find particle accelerations

```
1 root ← BUILD_OCTREE(particles)
2 for each [p_active, m_active] ∈ particles do
3     TRAVERSE_PASSIVE(p_active, m_active, root)
4 a_global ← ⟨0,0,0⟩
5 COLLAPSE_PASSIVE(root, a_global)
```

Implementing reverse Barnes-Hut validates the passive side of the Dual Hierarchy algorithm. Tests show that this algorithm is significantly less accurate than normal Barnes-Hut, given the same value of θ. This necessitates the use of higher multipole orders in order to meet the descent criterion. It also significantly underperforms the original Barnes-Hut, likely because of its poor memory access patterns. Because the passive tree is shallower than the active tree, Dual Hierarchy rarely reaches cases where a leaf of the active tree is reached first. This rare edge case is not handled by the reverse Barnes-Hut algorithm, instead, we decay to naive interactions between particles immediately.

4.2 Adaptation from t-SNE

Gravitational n-body has several differences from t-SNE, and these must be addressed with changes to the Dual Hierarchy algorithm. This section discusses each major difference in turn, alongside the changes which were made to address that difference.

4.2.1 Inverted Forces

The first difference is natural: our adapted algorithm must simulate gravity instead of the t-SNE repulsive force. This is simplified by the fact that the two forces behave similarly.

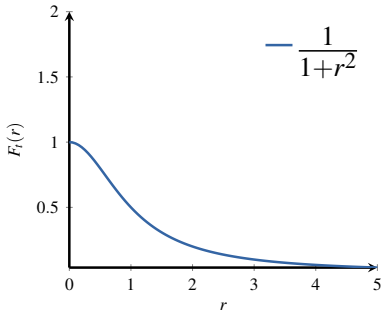
The magnitudes of the two forces are expressed in terms of r as shown in Fig. 4.3, where r is equivalent to $\|R\|$ and R is the vector connecting the source of the force to the sampling location. Both forces are applied along the direction of this vector.

While the t-SNE minimization force shown in Fig. 4.3a is repulsive and well-behaved near $r = 0$, the gravitational force in Fig. 4.3b is attractive, and becomes very large at small distances. A value ε is added to the denominator to prevent gravity from approaching infinity, but this value is typically kept very small, as it introduces inaccuracy at all distances.

It is important to note that the equation shown for t-SNE is not a complete expression of the t-SNE minimization force. There is also an attractive component, which is calculated separately and exclusively for elements that are close neighbors in the higher dimensional space.

In both equations, coefficients are omitted. For t-SNE the coefficient is a changing "momentum" which helps the embedding converge in fewer iterations. For gravity, the coefficient is Gm_1m_2, as given by the Newtonian equation for the force of gravity. In practice, we directly compute the acceleration of gravity using the coefficient Gm_1. In

(a) Repulsive t-SNE minimization force.



(b) Attractive gravitational force.

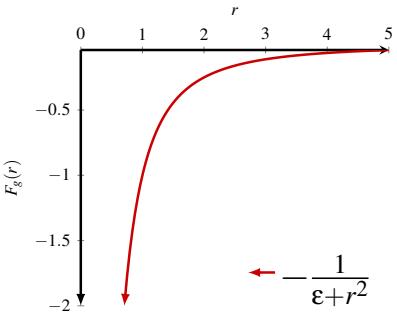


Figure 4.3: The equations for the magnitude of t-SNE and Gravitational forces.

the implementation, these coefficients can be applied after-the-fact, and do not affect the behavior of any approximation scheme. The m_1 coefficient does need to be applied to each force individually. This change will be discussed in Section 4.2.2.

The most important commonality between the two forces is that they both diminish quadratically at greater distances. This is the property that enables the use of summaries and tree algorithms for both problems.

4.2.2 Non-Uniform Masses

Unlike the particles in a t-SNE embedding, the point masses in an n-body do not all exert the same amount of force on one another. A particle with a greater mass produces a stronger gravitational field.

To handle this, particle masses are provided in the same way as the particle positions. The appropriate mass is included in each Particle-Particle interaction.

For node summaries, we substitute the particle count used in the original Dual-Tree method for a net mass value. This net mass is factored into Node-Particle and Node-Node interactions.

In practice, the AGORA dataset uses each particle to represent a cohort of stars, instead of a single star. Each cohort has the same mass, but the value is not 1.0. Correct handling of non-uniform masses must instead be tested with randomly generated data.

4.2.3 Strict Accuracy Requirements

The eventual convergence of the t-SNE embedding is robust against a high degree of inaccuracy, which allows for relatively large θ values. Generally, so long as computed forces bring particles closer to a high-quality configuration, correctness does not matter. Adaptive momentum and other gradient descent techniques help to further smooth out occasional major errors. Measures of quality for t-SNE are discussed in greater detail in Section 3.1.2.

n-body gravity is a chaotic dynamical system (Boccaletti and Pucacco [1998]). One implication of this is its extreme sensitivity to errors. The net force calculated on every particle must be accurate to a certain margin. If even a single particle experiences too much error, the entire system will quickly deviate from accurate results. The accuracy metrics used for cosmological simulations are therefore extremely strict. These metrics are discussed in detail in Section 3.4.

Degeneration to Direct Interactions

The original Dual Hierarchy approach produced a coarse field tree, and the forces in that tree were then distributed to the particles. For gravitational n-body it is necessary to degenerate to direct particle-particle interactions for very near-field interactions, where the field tree cannot provide the necessary accuracy.

Because of this change, the penalty for having a tree that is too shallow is very high. If one node contains 100,000 particles, near-field interactions must be computed for each particle individually. To resolve this, the tree is not built to a fixed depth and instead subdivided as necessary (up to a maximum). This may appear to have a large cost, but Section 5.4.1 shows that on the CPU, the additional tree-construction time is small for the scales tested.

Non-Projected Diagonal

The projected diagonal descent criterion described in Section 3.3.1 guarantees a maximum error in distance, but allows for some error in direction. For gravitational n-body, distance and direction are both important for accuracy. Attempts to apply the Projected-Diagonal descent criterion resulted in unbounded error, even for small values of θ and when high-order multipoles were used.

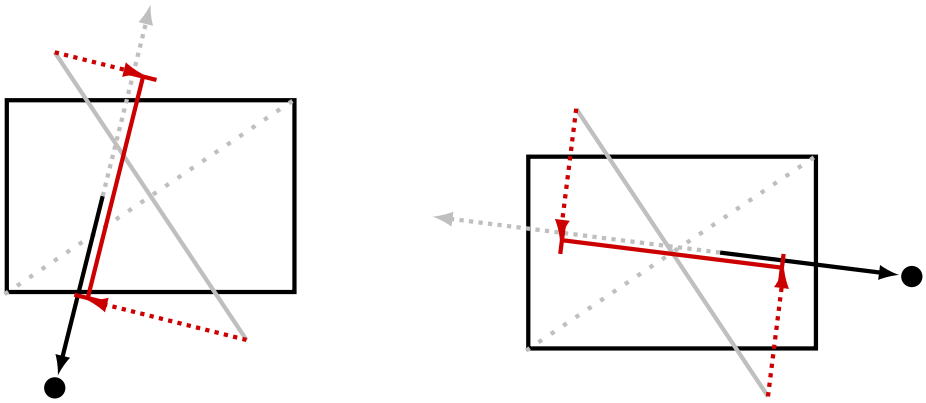


Figure 4.4: A pair of reverse projected diagonals (red) for interactions with a rectangular node. The original diagonal is shown as a dotted line, and the flipped diagonal is solid. Interactions along the *shorter* side of the node have longer reverse projected diagonals.

It is also possible to reverse the projected diagonal criterion by projecting onto a vector perpendicular to the connecting vector, instead of parallel. This can be seen in Fig. 4.4. This bounds directional error while allowing for more error in distance. This did not measurably improve accuracy over the original projected diagonal.

Our error metrics only care about the deviation from the correct force vector, as discussed in Section 3.4. They do not differentiate between errors in the magnitude of the force and errors in its direction. The force changes linearly with respect to differences in direction, and at medium range, the force of gravity declines approximately linearly with increases in distance. This means that gravitational error is effectively isotropic, which explains why the projected diagonal descent criterion is not beneficial.

Exclusion Region

For the dual traversal used by Dual Hierarchy for t-SNE, the descent criterion looks like the following:

$$\text{DESCENT\_CRITERION}(\text{node}_{\text{active}}, \text{node}_{\text{passive}}) = \begin{cases} \textbf{True} & \frac{\text{MAX}(S_{active\_node}, S_{passive\_node})}{\|p_{active\_node} - p_{passive\_node}\|} < \theta \\ \textbf{False} & \text{Otherwise} \end{cases} \tag{4.1}$$

The use of field nodes for gravity also requires a modification to this descent criterion. Because θ is measured relative to the center of mass, pathological cases can occur when a couple of conditions are met:

i. The center of mass of a node is on the far side of its bounding box (relative to the point the field is sampled).
ii. The node contains point masses on the near side of its bounding box.

When the center of mass is on the far side of the bounding box, the θ criterion allows the approximation to be used for sampling locations arbitrarily close to the near side. If there are point masses on the near side (small or few enough to affect the center of mass minimally), then the sampling location could be arbitrarily close to those masses. When computing the exact field, those nearby masses should dominate, but the approximation disregards them entirely. This allows for unbounded error.

This is a much larger issue for Gravitational n-body than for t-SNE, for reasons beyond the higher accuracy requirements. Because particles can have varied sizes, it is possible to have a very small particle that contributes a negligible amount to the center of mass. Moreover, the magnitude of the force F_G can become extremely large at close ranges, as seen in Fig. 4.3b. This means that even a very small mass can dominate the gravitational field nearby.

This issue was first discussed in the article "Skeletons from the Treecode Closet" (Salmon and Warren [1994]). It can be addressed directly with the addition of an "exclusion region", as shown in Fig. 4.5 (Springel et al. [2021]). This disallows sampling the field closer than half the side length of the node. The result is a backstop, preventing the pathological cases which θ fails to catch.

The exclusion region is integrated with the original descent criterion as shown in Eq. (4.2). The approximation is only valid when both the original criterion is satisfied and the exclusion region of the active node and the bounding box of the passive node are disjoint.

$$\begin{aligned} &\text{DESCENT\_CRITERION}(\text{node}_{\text{active}}, \text{node}_{\text{passive}}) \\ &= \begin{cases} \textbf{True} & (\frac{\text{MAX}(S_{active\_node}, S_{passive\_node})}{\|p_{active\_node} - p_{passive\_node}\|} < \theta) \\ & \wedge\ (\text{EXCLUSION}(\text{node}_{\text{active}}) \cap \text{node}_{\text{passive}} = \emptyset) \\ \textbf{False} & \text{Otherwise} \end{cases} \end{aligned} \tag{4.2}$$

To be used in the Dual Hierarchy approach, this must be adapted to work with the rectangular nodes of a Linear-Bounding Volume Hierarchy (BVH). Several solutions are explored in Appendix B, and ultimately the definition in Fig. 4.6 is selected.

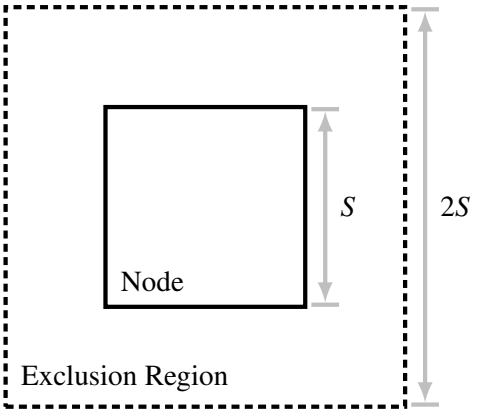


Figure 4.5: An octree node and the exclusion region which surrounds it.

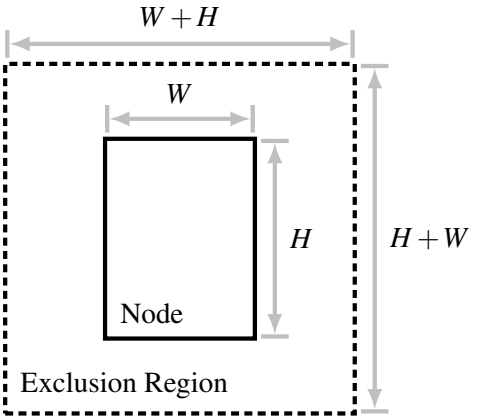


Figure 4.6: A rectangular Linear-BVH node and a square exclusion region that fits it optimally.

Note that for the special case where the node is square, this exclusion region formulation produces the same boundary as the original definition. In three dimensions, the exclusion region is not necessarily cubic, and its dimensions are defined by Eq. (4.3).

$$\begin{aligned} E_x &= S_x + \text{MAX}(S_y, S_z) \\ E_y &= S_y + \text{MAX}(S_x, S_z) \\ E_z &= S_z + \text{MAX}(S_x, S_y) \end{aligned} \tag{4.3}$$

Where E_x, E_y, E_z are the dimensions of the exclusion region and S_x, S_y, S_z are the dimensions of the node it encloses.

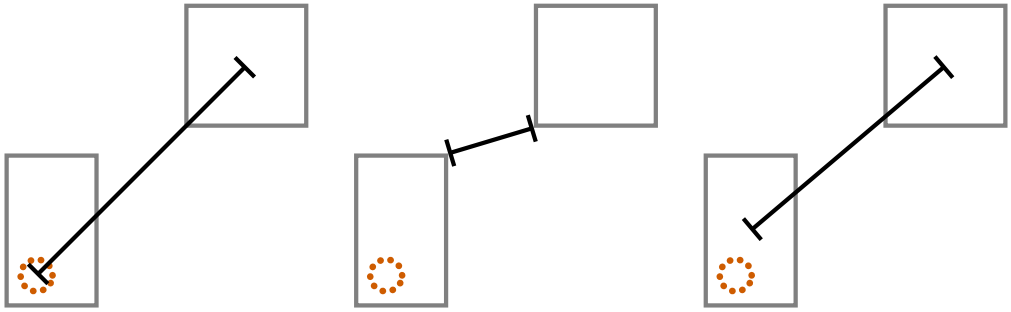
Separation

Traditional Barnes-Hut and FMM implementations measure distance based on the center-of-mass of the active node (and the center of the passive node, in the case of FMM), as shown in Fig. 4.7a. This makes intuitive sense, because the particles may be clustered far away from the center of the node, but this is the source of the edge case

which necessitates exclusion regions, as discussed in Section 4.2.3. Exclusion regions can be made unnecessary through the use of alternate measures of distance.

The Separation of a pair of modes is the minimum distance between their bounding boxes, as shown in Fig. 4.7b. It is an attractive choice because it is equivalent to the minimum possible distance between the nodes that they contain, making it a good measure of maximum error.

Separation is a much more strict measure than center-of-mass, especially for large nodes near the top of the tree. In Fig. 4.7, separation is much smaller than center-of-mass for the same nodes. Generally, higher average strictness can be accounted for with larger values of θ in order to produce the same error, but differences in how that strictness is distributed can produce significant differences in performance.



(a) Center-of-mass (b) Separation (c) Center-to-center

Figure 4.7: Alternative measures of distance for a pair of nodes.

Center-to-Center Distance

We can also eliminate exclusion regions if we use center-to-particle or center-to-center distance as shown in Fig. 4.7c, we prevent the issue of a shifted center of mass.

This is an appropriate solution only for a Linear-BHV. Because the shape of an Octree node is fully determined by its place in the overall tree, the points it contains may be clustered far away from the center. A Linear-BVH node always tightly fits the nodes it contains, meaning that while the center of mass may be far from the center, the diagonal length is always an accurate indicator for the maximum deviation relative to the center. When calculating forces we need to use the center of mass, but when deciding whether to use an approximation, it is the maximum allowable deviation that matters.

Descent Criterion Selection

Fig. 4.8 shows benchmark results for each of these variants of the Linear-BVH Barnes-Hut algorithm. In each case, θ is adjusted so that they achieve equivalent accuracy. The first two versions use the optimal exclusion region determined in Section 4.2.3, while the other two use no exclusion region. The third uses the separation criterion described in Section 4.2.3, and the fourth uses center-to-center distance as described in Section 4.2.3.

Tests were done on the AGORA-low dataset, with quadrupole node summaries. Fig. 4.8a shows the number of each type of interaction as a percentage of the number

of interactions the naive approach would be required to compute. Because this is an active-tree method, only particle-particle and node-particle interactions are done. Notice how only a small number of node-particle approximations eliminate over 90% of naive interactions.

Fig. 4.8b shows the resulting iteration times of the same tests. These results are strongly correlated with the interaction counts, algorithms that perform a greater number of interactions take longer to run. One discrepancy is that the *side-length over distance* algorithm is slower than we might expect from its interaction count. Looking closely at Fig. 4.8a, it performs more node-particle interactions than the *diagonal over distance* algorithm. This likely accounts for the discrepancy because quadrupole node-particle interactions are significantly more expensive to compute than particle-particle interactions.

Benchmarking showed that the center-to-center approach outperformed all approaches which required exclusion regions. It has the best approximation ratio of all descent criteria, and this translates to the fastest runtime.

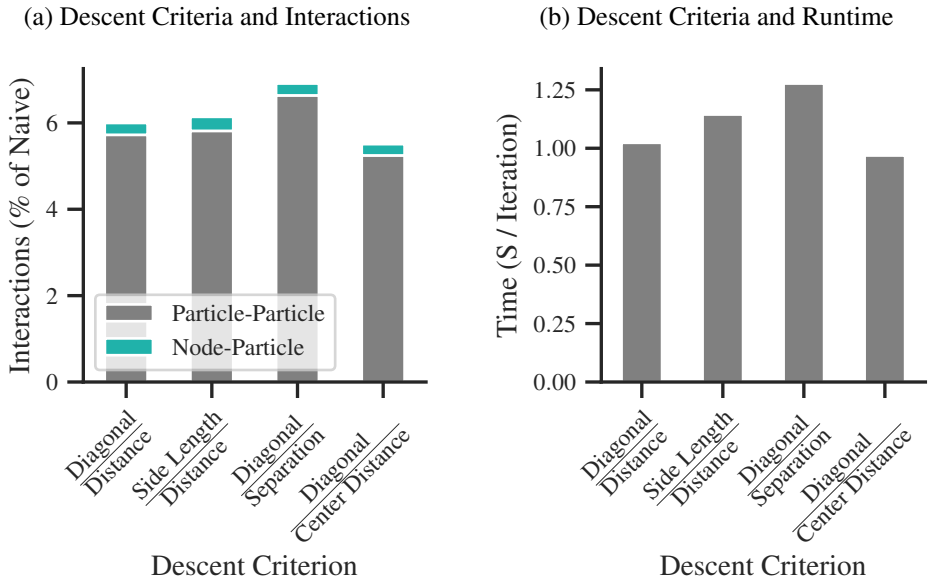


Figure 4.8: The effects of descent criterion choice on Linear-BVH solver performance

Multipole Decomposition

The previous changes all improve accuracy, but combined they still only allow for appropriately bounded error when θ is relatively low. In order to meet our strict accuracy requirements, multipole moments and fields (Section 3.5) are critical. The optimal multipole order for active and passive nodes is determined in Section 5.1.2.

4.2.4 Higher Dynamic Range

Cosmological *n*-body scenarios have higher dynamic ranges than typical t-SNE minimizations. This means that datasets like AGORA can often contain clusters where

a very large number of particles occupies a small space, combined with very large regions which only contain a handful of particles.

Combined with values for n which are orders of magnitude higher, this means that the Linear-BVH becomes saturated for even small test cases. Tightly clustered particles can all be assigned the same Morton code, and as a result, they are all assigned to the same node in the tree. For a 32-bit Linear-BVH, we interleave 10 bits of each axis, for a maximum tree depth of 30.

The solution is to expand our Linear-BVH and use 64-bit Morton codes. These Morton codes are produced by interleaving 21 bits from each axis, for an expanded maximum tree depth of 63. This has only a minor performance penalty on Linear-BVH construction times, likely due to the extra depth more than the increased code size.

4.3 Implementation

The test implementation was written in C++20, with heavy use of template metaprogramming to enable the reuse of algorithms. This is important because it means the algorithms under study are comparable, sharing the vast majority of their implementation.

A number of C++ libraries were used in development. Visualization is done using an OpenGL wrapper provided by the Magnum graphics engine. Multithreading was accomplished using the Intel Threading Building Blocks library (TBB). Vector math was done using the OpenGL Mathematics Library (GLM). And a Radix-sort implementation from Boost was used, made parallel using TBB.

Plotting was done in Python, using Pandas to load the CSV files produced by tests, and Matplotlib through Seaborn to produce the final plots.

4.3.1 Trees

The tree type is split into several parts, which can be combined using templates. This modular design makes it simpler to perform a fair comparison of different trees; because the majority of the code is the same, implementation quality is less of a factor.

The **Node** type provides functionality related to the topology and memory layout of the tree, such as the number of children and how those children are allocated. It also provides the logic which drives the top-down portion of construction, deciding how particles are partitioned between the children.

The **Summary** type provides functionality relevant to the solver, this can be much more varied than the Node type. Examples range from a center-of-mass summary which is used on the active side of the simple Barnes-Hut solver, to a multipole field summary which is used on the passive side of the gravity calculations in some other solvers. The summary type also has responsibility for the bottom-up portion of tree construction. A summary implementation must provide a method that summarizes a list of particles, and another that summarizes a list of nodes.

A **split-criterion** functor determines the overall shape of the tree. It decides whether any given node needs to be split. The simplest split criterion returns **True** for any node which contains more than one particle. Typically, we use a split criterion that provides a maximum node size, e.g. a node is split IFF it contains more than 32

particles. This helps to prevent singleton chains, situations where a pair of particles are extremely close together so that the tree must be excessively deep in order to divide them into their own nodes. This aspect of tree construction is discussed in more detail in Section 4.5.2.

Tree construction is implemented generically by REFINE(root), where REFINE() works as shown in Algorithm 9, and the functions SPLIT(), MERGE(), and SUMMARIZE(), are all specific to the type of tree being constructed. This implementation also works for efficiently re-constructing existing trees without excess memory allocation, merging nodes that are no longer needed.

Algorithm 9 A generic recursive algorithm for the construction of any type of tree.

```
1 procedure REFINE(node)
2     if SPLIT_CRITERION(node) then
3         SPLIT(node)
4         for each child ∈ node do
5             REFINE(child)
6     else
7         MERGE(node)
8     SUMMARIZE(node)
```

4.3.2 Tensors & Multipoles

While C++ libraries like Eigen are good for working with large, sparse, and variable-sized tensors, there are limited options for working with smaller, fixed-size, symmetric tensors like those which make up multipoles, as described in Section 3.5. To achieve good performance, it was necessary to implement a specialized 3^R Tensor library, and template-metaprogramming is well suited for this task.

The elements of a symmetric tensor are stored using an array. As discussed, because the symmetric tensor is symmetric many values are redundant, especially for higher-rank tensors. We only store the unique elements. Access is done by index, with `tensor.get<X, Y, Y>()` providing access to the element at the coordinate $(0,1,1)$ within a rank-3 tensor. The indices are template parameters so that they can be converted to a 1-dimensional index in the underlying packed array at compile time, eliminating overhead for access at runtime. Because the tensor is symmetric, `tensor.get<Y, X, Y>()` should produce the same value as `tensor.get<X, Y, Y>()`. To accomplish this, we put the index in a canonical form before converting it to a flat index. In C++20, we can do this using `constexpr`, applying `std::sort` at compile-time.

Operations on tensors often involve nested loops, producing values for each index of a tensor based on values of another tensor. These are simple to implement but may be difficult for the compiler to fully optimize. In order to ensure SIMD instructions are used wherever possible, it is best to unroll these loops. Doing this manually is possible, and this is actually how GADGET-4 implements its tensor types, but it quickly becomes unwieldy for higher dimensional tensors. The cross-product of a pair of rank-3 tensors can easily take dozens of lines of code, and this only gets worse in higher dimensions. In order to experiment with arbitrarily high dimensional tensors, we want to make the rank generic, without sacrificing our loop-unrolling optimization. The solution is to use fold expressions as shown in Code fragment 1.

Code fragment 1 C++ implementation of the outer product of two tensors

```
template<std::size_t Order, std::size_t OtherOrder>
SymmetricTensor3<Order - OtherOrder> operator*(
    const SymmetricTensor3<Order> &lhs,
    const SymmetricTensor3<OtherOrder> &rhs
) {
    constexpr std::size_t ProductOrder = Order - OtherOrder;
    SymmetricTensor3<ProductOrder> product{};
    [&]<std::size_t... I>(std::index_sequence<I...>) {((

        // The following expression is applied for each value of I
        product.template get<(
            head<lexicographicalIndex<I>(), ProductOrder>()
        )>() +=
                lhs.template get<(
                    lexicographicalIndex<I>()
                )>() *
                rhs.template get<(
                    tail<lexicographicalIndex<I>(), OtherOrder>()
                )>()

    ), ...); }(std::make_index_sequence<NumValues>());
    return product;
}
```

Fold expressions require rather baroque and hard-to-interpret code, but the underlying concept here is straightforward: We are unrolling a loop for each `I` in the range from 0 to the number of indices in the tensor (3^{Order}). `lexicographicalIndex<I>()` converts `I` to a multidimensional index in the format `i, j, k`, and then `head<>()` and `tail<>()` extract subsets of that dimensional index. For each value of `i, j, k` in an order-3 tensor, the unrolled loop evaluates the following:
`product.get<i, j>() += lhs.get<i, j, k>() * rhs.get<j, k>()`.

Moment and field multipoles were defined using templated sequences of tensors. A multipole moment contains a scalar mass component, a center vector, and a tuple of SymmetricTensor<R> types, with R ranging from 2 (a 3×3 matrix) to the rank of the multipole (for example, 4 for a Hexadecupole). A multipole field is defined in a similar way, but it does not contain a scalar component and its vector component represents the uniform component of the field; its center position is implied by its location within the tree. A collection of useful operators is also defined for the multipole types, typically implemented using the operators of their constituent tensors.

4.3.3 Physics Rules

The core of the *n*-body solver is the kernel that implements the gravity equation. Even in solvers which perform a great deal of approximation, the physics kernel is an especially "hot" section of the code. For complex solvers, it becomes necessary to implement several of these kernels, because node-node, node-particle, and particle-particle

interactions are each handled differently. In this implementation, the kernels are provided as methods of a "Physics rule" type.

This provides significant flexibility. The gravitational physics rule could easily be replaced by one which calculates electrostatic forces, or even the repulsive t-SNE minimization force.

This flexibility also made invasive tracking straightforward, allowing for a richer suite of benchmarks. By adding tracking to the physics rule itself, we can produce a detailed log of the properties of each interaction, including the properties of all the nodes and particles involved. This is done using a template type that wraps the underlying physics rule. Each time one of the interaction methods of the wrapper type is invoked, it updates the log before delegating to the underlying rule.

One of these wrapper types saves all the details of each interaction as a new line in a CSV file. This is naturally extremely expensive, made more so by the fact that the file must be protected by a mutex, effectively forcing all interactions to happen sequentially. This tracker is only used for small-scale tests which look at the relative sizes of nodes, such as in Section 5.4.2.

A faster but less detailed tracking wrapper was also created, which only counts the different types of interactions by incrementing atomic integers. The approximation ratio of an algorithm (Section 4.5.3) can be measured using the simpler tracker.

For all timing-sensitive benchmarks, no invasive tracking is performed, because even the simpler tracker has a significant performance penalty. Luckily, all the tested algorithms have deterministic behavior, so the logs from tracked benchmarks can be cross-referenced with times from un-tracked benchmarks.

4.3.4 Solvers

The core of any n-body algorithm is the approach it uses to compute the accelerations of each particle, every iteration. Often it can help performance if some state is retained between iterations, for example by updating a tree rather than building it from scratch. To enable this, each algorithm is a method of a `Solver` type which is bound to a specific scenario and rule.

We define one Solver type for each of the algorithms tested. Aside from the naive solver, each of these is defined as a template instantiation. For example, the Barnes-Hut solver declaration is shown in Code fragment 2. This solver is based on `ActiveTreeSolver<>`, where the type of tree used and the descent criterion are both template parameters. This means we can easily produce a new algorithm by substituting a Linear-BVH for the Octree or swapping out the descent criterion for one which does not enforce an exclusion region.

Code fragment 2 Barnes-Hut solver type declaration

```
template<RuleType Rule>
using QuadrupoleBarnesHutSolver = ActiveTreeSolver<
        MultipoleActiveOctree<2>,
        Descent::SideLengthOverDistance,
        Rule
>;
```

The adapted dual hierarchy method is implemented as shown in Code fragment 3. It is a dual-tree solver, where the active (mass) tree is a Linear-BVH and the passive (acceleration) tree is an Octree. It uses the `DiagonalOverCenterDistance` descent criterion, meaning that it will use an approximation when $\theta < \frac{\text{Maximum diagonal}}{\text{Center-to-center distance}}$.

Code fragment 3 Adapted dual hierarchy solver type declaration

```
template<RuleType Rule>
using QuadrupoleImplicitDualHeirarchySolver = ImplicitDualTreeSolver<
        MultipoleActiveLinearBVH<2>,
        MultipoleImplicitPassiveOctree<3>,
        Descent::DiagonalOverCenterDistance,
        Rule
>;
```

Each Solver implementation is templated on the Physics Rule type, so that we can substitute different rules, including the wrapper rules which track interactions. It is important that this is a template and not a parameter; we want the compiler to inline our physics kernels for the best performance.

Most benchmarking code is templated on the Solver type so that we can drop-in different solver implementations to test them.

4.4 Optimizations

Once we have an algorithm capable of meeting our accuracy requirements, we can work to improve its performance through optimization. In section Section 4.4.1 we explore changes to the descent method for increased approximation and better thread occupancy. Next, we reduce the memory requirements by not storing the net accelerations in the passive tree in Section 4.4.2.

4.4.1 Smarter Descent

Performing more approximations will generally increase the mean error. So long as our condition for performing an approximation is sound, adding more valid approximations should not increase our theoretical maximum error. In practice, the measured maximum error does slowly increase, because the original algorithm which performed fewer approximations may not have run into the same edge cases. That does not mean that it guaranteed that those edge cases would never occur – the lower error may have only been incidental.

As a consequence, we should expect increasing valid approximations to raise the average error while keeping the maximum error bounded. It helps to imagine a theoretically "optimal" algorithm that performs the most aggressive approximations allowable. This algorithm, if it existed, would have a mean error equivalent to its maximum error. It would also compute the fewest interactions needed to meet the accuracy requirement. Our aim is to bring the Dual Hierarchy algorithm closer to this theoretical optimum by opening up more opportunities for approximations. This can be done by adding more complex behavior to the descent algorithm.

Lockstep Descent

A simple implementation of lockstep dual traversal is used by the original FMM algorithm, shown in Algorithm 4. This procedure recursively computes the acceleration of $\text{node}_{\text{passive}}$ and its contents based on $\text{node}_{\text{active}}$ and its contents. Because $\text{node}_{\text{active}}$ and $\text{node}_{\text{passive}}$ do not need to be parts of the same tree, this same procedure also implements dual-tree dual-traversal algorithms like Dual Hierarchy, without any changes.

This is not necessarily optimal for the purpose, however. With dual-traversal of a single octree, as in FMM, descending both the active node and the passive node produces interactions between children that are the same size. When dual-traversal is performed over two different trees, this is not guaranteed to be the case.

For the combination of Linear-BVH and Octree, this is a particularly large issue. Linear-BVH nodes are not of uniform size, and the active tree has a splitting ratio of 2 instead of 8. This means that in some cases, descending both nodes can decrease the size of the active nodes by a much greater factor than the passive nodes, and in other cases, the reverse can be true.

In practice, the multipole field summary used on the passive side is less accurate than the multipole, and if we allow the passive node to be half the size of the active node we can use higher values of θ and see better performance.

Adaptive Descent

Adaptive Descent attempts to solve this issue by not descending the smaller of the two nodes if the difference is large. This means that the algorithm is always attempting to even out the sizes of the nodes, and an imbalance is less likely to form.

An implementation of such an algorithm is shown in Algorithm 10. Here, DESCENT_CRITERION is modified to return a "recommendation": APPROXIMATE when the nodes are far apart, DESCEND_BOTH when the ratio between the nodes is around our target, and DESCEND_ACTIVE or DESCEND_PASSIVE when one node is too small. Accommodating leaves on one side of the tree or the other adds a number of edge cases, and the implementation quickly becomes unwieldy. Notice how we never decay all the way to particle-particle interactions; instead, the nodes are passed to a passive- or active-traversal, and if both nodes are leaves that traversal will perform the direct interactions.

Algorithm 10 A recursive implementation of Adaptive Dual-Tree Traversal

```
1  procedure TRAVERSE_ADAPTIVE(node_active, node_passive)
2      recommendation ← DESCENT_CRITERION(node_active, node_passive)
3      if recommendation = APPROXIMATE then
4          GRAVITY(node_active, node_passive)
5      else if recommendation = DESCEND_ACTIVE then
6          if IS_LEAF(node_active) then
7              for each [p_active, m_active] ∈ node_active do
8                  TRAVERSE_PASSIVE(p_active, m_active, node_passive)
9          else
10             for each child_active ∈ node_active do
11                 TRAVERSE_ADAPTIVE(child_active, node_passive)
12     else if recommendation = DESCEND_PASSIVE then
13         if IS_LEAF(node_passive) then
14             for each [p_passive, a_passive] ∈ node_passive do
15                 a_passive ← a_passive + TRAVERSE_ACTIVE(node_active, p_passive)
16         else
17             for each child_passive ∈ node_passive do
18                 TRAVERSE_ADAPTIVE(node_active, child_passive)
19     else
20         if IS_LEAF(node_active) then
21             for each [p_active, m_active] ∈ node_active do
22                 TRAVERSE_PASSIVE(p_active, m_active, node_passive)
23         else if IS_LEAF(node_passive) then
24             for each [p_passive, a_passive] ∈ node_passive do
25                 a_passive ← a_passive + TRAVERSE_ACTIVE(node_active, p_passive)
26         else
27             for each child_passive ∈ node_passive do
28                 for each child_active ∈ node_active do
29                     TRAVERSE_ADAPTIVE(child_active, child_passive)
```

Unfortunately, adaptive descent does not produce a meaningful performance benefit. The increase in approximation power is small and outweighed by the slight increase in error resulting in lower θ and the major increase in descent logic complexity. The cost of the divergent code increases further when implementing an Implicit Field variant of this algorithm (discussed in Section 4.4.2).

Balanced-Lockstep Descent

The investment in smarter descent logic pays off when attempting to increase our CPU core occupancy. In Algorithm 11, we implement an iterative algorithm that uses the recommendation from DESCENT_CRITERION to descend an active node until it reaches nodes of an appropriate size to interact with a passive node. Once it has a balanced node list, it switches to conventional Lockstep descent.

Algorithm 11 An iterative implementation of Balanced-Lockstep Traversal

```
1  procedure TRAVERSE_BALANCED_LOCKSTEP(root_active, node_passive)
2      queue ← [root_active]
3      appropriately_sized_nodes ← []
4      while SIZE(queue) > 0 do
5          node_active ← POP(queue)
6          recommendation ← DESCENT_CRITERION(node_active, node_passive)
7          if recommendation ≠ DESCEND_ACTIVE ∨ IS_LEAF(node_active) then
8              PUSH(appropriately_sized_nodes, node_active)
9          else
10             for each child_active ∈ node_active do
11                 PUSH(queue, child_active)
12     for each node_active ∈ appropriately_sized_nodes do
13         TRAVERSE_LOCKSTEP(node_active, node_passive)
```

The obvious approach to parallelism is to create a job for each passive node at a fixed depth, which performs a traversal interacting that node with every active node at another fixed depth. This can result in low occupancy in high-dynamic-range datasets because some of these nodes will contain far more particles than others. With the balanced-lockstep descent approach, we can instead split up the work by dividing the most populous node until we have the desired number of starting nodes. This gives us a list of passive nodes with various dimensions, each of which has a similar number of contained particles. We then spawn a job interacting each with the active root, and TRAVERSE_BALANCED_LOCKSTEP() descends the root node to produce good starting points.

4.4.2 Implicit Field-Tree

In the simple dual-traversal algorithm discussed so far, we store the field multipoles explicitly in each node of a dedicated Octree. The ability to view these values after the traversal has been completed but before the collapse is useful for debugging. It allows analysis and visualizations like those in Section 5.1.1.

Unfortunately, storing and writing to a tree full of multipoles is expensive. Each quadrupole requires 24 bytes to store, and this gets worse for octupoles and higher. Not only does it take time to write to each multipole in memory, larger multipoles mean field node addresses must be spaced further apart, increasing the cache miss-rate.

We are traversing the passive tree twice – once to write to each multipole field, and once to collapse the fields. We can combine these traversals and keep track of the net acceleration in the current branch instead. This means that each thread only needs to store a single multipole, and updates to this multipole never require modifying heap data.

To understand this change, it helps to first examine a simpler version that implements reverse-Barnes-Hut. Algorithm 12 combines the passive-tree traversal (Algorithm 6) and the subsequent collapse (Algorithm 7) into a single recursive function.

Algorithm 12 A recursive implementation of Implicit Passive-tree Traversal.

```
1  procedure TRAVERSE_IMPLICIT_PASSIVE(active_particles, node_passive, A_node)
2      p_node := node_passive
3      (far_particles, near_particles) ←
           PARTITION_BY_DESCENT_CRITERION(active_particles, node_passive)
4      for each [p_active, m_active] ∈ node_active do
5          A_node ← A_node + GRAVITY(p_active, m_active, p_node)
6      if IS_LEAF(node_passive) then
7          for each [p_passive, a_passive] ∈ node_passive do
8              for each [p_active, m_active] ∈ node_active do
9                  a_passive ← a_passive + GRAVITY(p_active, m_active, p_passive)
10         for each [p_passive, a_passive] ∈ node_passive do
11             a_passive ← a_passive + EVALUATED_AT_OFFSET(A_node, p_passive − p_node)
12     else
13         for each child_passive ∈ node_passive do
14             TRAVERSE_IMPLICIT_PASSIVE(near_particles, child_passive, A_node)

15 procedure PARTITION_BY_DESCENT_CRITERION(active_particles, node_passive)
16     Partitions the list of particles based on how they relate to the passive node.
17     Elided for brevity.
18     return (far_particles, near_particles)
```

Each $\text{node}_{\text{passive}}$ in the implicit traversal only contains information about the tree structure and no multipole. Instead, A_{node} is a multipole field representing the local acceleration due to forces from faraway particles. Because it is passed down recursively, when a leaf node is reached, A_{node} includes the sum of the forces which were calculated at every level of the tree. At this point, the net acceleration is distributed to all local particles, and local forces are also calculated directly.

PARTITION_BY_DESCENT_CRITERION() splits the list based on whether the particles are far enough away to use $\text{node}_{\text{passive}}$ for force approximations (according to the same DESCENT_CRITERION() used in other traversals. In the implementation, this is done by rearranging the list itself and returning iterators, so that no new memory needs to be allocated. These iterators are passed to each branch of the descent in turn, so the same list of particles is repeatedly rearranged.

This precludes any parallelism beyond the top-level task assignment already discussed in Section 4.4.1, but with good load balancing, that does not prevent high occupancy. This technique is unlikely to be adapted for the GPU in its current form, but that does not necessarily imply that an implicit field hierarchy is impossible on the GPU.

Expanding the algorithm to dual-tree comes with one additional challenge: In the original algorithm, each field node is written to many times, as it receives forces from many sources – both from other nodes and directly from particles. Before we can descend the tree, we need our local acceleration to reflect all the long-range forces on the current node. In other words, the descent needs to be done breadth-first.

This requires major changes to the control flow of the algorithm, as seen in Algorithm 13. The traversal can still be done recursively, but we cannot split into a separate branch for each active node a passive node interacts with. The local acceleration must incorporate *all* of the far active nodes before that passive node can be further descended. To accomplish this, we have an "outer loop" in which the passive

tree is descended depth-first and an "inner loop" in which the active tree is descended breadth-first.

Algorithm 13 A recursive implementation of Implicit Dual-tree Traversal.

```
1  procedure TRAVERSE_IMPLICIT_DUAL_TREE(active_nodes, node_passive, A_node)
2      p_node := node_passive
3      if SIZE(active_nodes) = 0 then
4          for each [p_passive, a_passive] ∈ node_passive do
5              a_passive ← a_passive + EVALUATED_AT_OFFSET(A_node, p_passive − p_node)
6      else
7          (far_nodes, leaf_nodes, non_leaf_nodes) ←
               PARTITION_BY_DESCENT_CRITERION(active_nodes, node_passive)
8          for each node_active ∈ far_nodes do
9              A_node ← A_node+GRAVITY(node_active, p_node)
10         if IS_LEAF(node_passive) then
11             for each node_active ∈ leaf_nodes do
12                 for each [p_passive, a_passive] ∈ node_passive do
13                     for each [p_active, m_active] ∈ node_active do
14                         a_passive ← a_passive + GRAVITY(p_active, m_active, p_passive)
15             for each [p_passive, a_passive] ∈ node_passive do
16                 for each node_active ∈ non_leaf_nodes do
17                     a_passive ← a_passive+ TRAVERSE_ACTIVE(node_active, p_passive)
18                 a_passive ← a_passive
                               + EVALUATED_AT_OFFSET(A_node, p_passive − p_node)
19         else
20             for each node_active ∈ leaf_nodes do
21                 for each [p_active, m_active] ∈ node_active do
22                     TRAVERSE_IMPLICIT_PASSIVE(p_active, m_active, node_passive)
23             child_nodes ← []
24             for each node_active ∈ non_leaf_nodes do
25                 for each child_active ∈ node_active do
26                     APPEND(child_nodes, child_active)
27             for each child_passive ∈ node_passive do
28                 p_child := child_passive
29                 A_child ← EVALUATED_AT_OFFSET(A_node, p_child − p_node)
30                 TRAVERSE_IMPLICIT_DUAL_TREE(
                       child_nodes, child_passive, A_child
                   )

31 procedure PARTITION_BY_DESCENT_CRITERION(active_nodes, node_passive)
32     Partitions the list of active nodes into several groups
33     based on how they relate to the passive node.
34     Implemented as a Dutch-flag sort, elided for brevity.
35     return (far_nodes, leaf_nodes, non_leaf_nodes)
```

In order to address active nodes in breadth-first order, we need to maintain a list of relevant nearby nodes for each step of the passive tree descent. In Algorithm 12 we could rearrange one large list of all particles, but because we will be adding the children of each node as we descend, this isn't an option, and we'll need to provide a different list for each level. The size of these lists depends on θ, the type of tree, and

the distribution of particles; it appears roughly constant with respect to n. For FMM at appropriate θ to achieve constitutional error, it is typically in the low thousands. For the binary active tree of the Dual Hierarchy algorithm, this number is smaller, typically in the low hundreds. We have one list for each level of the (octree) passive tree, meaning that the total memory usage grows logarithmically with n.

4.5 Tuning & Hyperparameter Selection

This section explains how the algorithms tested in Chapter 5 were configured for the best performance. Section 4.5.1 explains how θ was selected in order to meet accuracy requirements with the lowest possible runtime, and Section 4.5.3 describes a metric that can be used to directly measure the efficacy of changes to an algorithm.

4.5.1 Selecting an Appropriate Value for θ

As discussed in Section 3.4, it is critical to choose an appropriate value of θ in order to minimize computation time while satisfying our accuracy requirements. Compute time decreases as θ increases, as a higher value of θ will typically result in fewer interactions being computed. Accuracy also decreases roughly monotonically with θ, though there is a small amount of noise because small changes to which approximations are made can have unpredictable effects on the maximum error.

Because the accuracy-performance trade-off is well-characterized, we can reliably find the optimal value of θ using a binary search. All solvers have θ values in the range $\theta_{\text{MIN}} = 0.1$ to $\theta_{\text{MAX}} = 0.9$, so that is used as the initial search range. The search ends when $\theta_{\text{MAX}} - \theta_{\text{MIN}} < 0.005$, and the lower value is used, ensuring that the accuracy criterion is met.

4.5.2 Selecting Maximum Leaf Node Sizes

Node-node and node-particle interactions are considerably more computationally expensive than particle-particle interactions, particularly when multipoles are involved. Not only does each quadrupole-octupole interaction require more operations, a group of these interactions is less likely to be vectorized by the compiler than the direct interactions. As such, it does not make sense to perform approximate interactions involving nodes when they only eliminate a small number of particle-particle interactions.

This leads us to incorporate a *maximum leaf size* hyperparameter. Instead of containing a single particle, each leaf node can contain a small list of particles. This also has the benefit of improving tree-quality by preventing singleton chains and reducing aspect ratios in a Linear-BVH.

This new hyperparameter must be tuned carefully. We performed benchmarks on both random and AGORA datasets, setting a maximum of 8, 16, 32, 64, 128, 256, and 512 particles per node during construction. A maximum size of 64 reliably performed best, indicating that a node-particle interaction is approximately as expensive as 64 particle-particle interactions.

For the adapted Dual Hierarchy algorithm, we want to reduce the likelihood of decaying to passive-tree traversals. The original Dual Hierarchy algorithm uses a fixed,

shallow, depth for the passive tree, but that was not an option for the high-dynamic-range datasets of gravitational n-body. Instead, we increase the maximum leaf size of the passive tree. Testing with 64, 128, 256, and 512 particles, we found the best performance when allowing up to 128 particles per node. This comes with the added benefit of reducing the average depth of the tree, resulting in shorter construction times.

4.5.3 Approximation Ratio

The Approximation Ratio R_A of an algorithm that solves the n-body problem is given by the total number of interactions it computes divided by the total number of directional particle pairs (equivalent to N^2). Particle-particle, particle-node, node-particle, and node-node interactions are all included in the numerator, though for not every algorithm performs each type of interaction.

$$R_A = \frac{I_{pp} + I_{pn} + I_{np} + I_{nn}}{N^2} \tag{4.4}$$

The naive algorithm computes every particle-particle interaction direction, so $I_{pp} = N^2$, and $R_A = 1.0$. A basic improvement to the naive algorithm takes advantage of the symmetry of the gravitational equation, $F_g(p_a, p_b) = -F_g(p_b, p_a)$, calculating each force only once for both directions. With this change, $I_{pp} = \frac{N^2}{2}$, and $R_A = 0.5$.

Table 4.1: Interaction counts for the Barnes-Hut algorithm ($\theta = 0.5$) on two different datasets of the same size.

	AGORA 112,500	Uniform 112,500
Particle-Particle	247,526,334	118,617,417
Node-Particle	56,023,677	43,381,661
Approximation Ratio	0.0239842 (41.7×)	0.0127999 (78.1×)

The Approximation Ratio for spatial-hierarchy-based algorithms can vary significantly based on the configuration and scenario. In Table 4.1, the Barnes-Hut algorithm is applied for both a uniform distribution of particles and the much more structured AGORA dataset. The uniform dataset produces a more balanced tree, allowing for more opportunities for approximation. This significantly reduces the number of interactions computed. Note that not all node-particle interactions are equally beneficial: the uniform dataset actually results in fewer, but each one eliminates more particle-particle interactions because it is done higher in the tree.

Chapter 5

Results

To understand the behavior of the Dual Hierarchy algorithm and determine how it compares to the state-of-the-art for gravitational *n*-body, we apply a suite of benchmarks and tests.

Interaction tracking benchmarks such as those described in Section 4.3.3 have deterministic results, and can only be performed on smaller datasets. The majority of these tests were performed on a 2022 MacBook Pro with an Apple M1 Pro ARM processor and 16 GB of memory. This processor features 8 total cores, with a heterogeneous "big.LITTLE" design. This means that the cores are divided between 6 high-power *performance* cores and 2 low-power *efficiency* cores.

To ensure that the unusual instruction set architecture of the M1 chip doesn't affect the results of larger-scale time-sensitive benchmarks, these were also performed on a contemporary x86-based processor. Tests were run across several identical nodes, each equipped with an Intel Xeon E5 2680 v4. Turbo mode was disabled for a consistent clock frequency of 2.4 GHz. Hyperthreading was disabled and only 8 physical cores were used for consistency with the M1 chip, and so that the load-balancing scheme could be used without re-calibration. Memory usage was not a limiting factor and was capped at 6GB per node for all tests. Batch configuration was set so that each benchmark would have exclusive access to the node, and would not be interrupted or transferred to another node mid-run.

Unless otherwise stated, θ for each solver is selected in order to meet the 0.5% constitutional error threshold on the AGORA-LOW dataset, using the process defined in Section 4.5.1.

In Section 5.1, we examine the effects of using different multipole orders in order to determine the optimal choice for performance, and also to understand how they fit into the tree structure. In Sections 5.2 and 5.3, we show that while the dual hierarchy algorithm is faster than many competing algorithms, it does not outperform the state of the art. In Section 5.4, we evaluate several theories for why it under-performs, including additional tree construction time (Section 5.4.1), mismatched node-sizes (Section 5.4.2), and poor memory access patterns (Section 5.4.3).

In plots, the implementation of an algorithm is referred to as a "Solver". The Barnes-Hut solver is abbreviated as BH, the Linear-BVH variant of Barnes-Hut is abbreviated as LBVH, the Fast Multipole method is abbreviated as FMM, and the dual hierarchy solver adapted from van de Ruit et al. [2022] is abbreviated as MVDR. For tabular data, the best result is always shown in **bold**.

5.1 Multipoles and Acceleration Fields

In Section 5.1.1, we look at how the use of quadrupoles and higher-order multipoles affect the field represented by a tree, in order to understand why the use of multipoles on the field side is so critical to accuracy. In Section 5.1.2 we measure the accuracy and performance of several multipole moment-field combinations and determine which configuration provides the best trade-off for each solver. This combination is then used for all future benchmarks.

5.1.1 Multipoles and Trees

Tests during development showed that multipole field representations were critical in order to meet accuracy metrics. The original center-of-mass based Barnes-Hut method produces results with relatively high error, which can still be bounded using lower values for θ. The reversed Barnes-Hut algorithm explored in Section 4.1.2 was originally tested with a vector representation for node acceleration instead of a quadrupole. This version has unbounded error and does not meet even very loose accuracy requirements unless θ is reduced to a point where no approximations are performed. We can see why quadrupole accelerations are so much more critical than quadrupole moments if we examine the field that they encode when used as part of a tree.

A center-of-mass representation produces a physically plausible field, consistent with the field produced by fewer, heavier bodies. In contrast, field tree representations are physically implausible by default. Using a vector to represent the field across each node results in large regions of uniform acceleration with discontinuous borders between nodes. Figure 5.1 shows a 2-dimensional slice of the gravitational field, sampled across a simulation domain. The discontinuities in the field are most visible in the less dense regions, where nodes are larger.

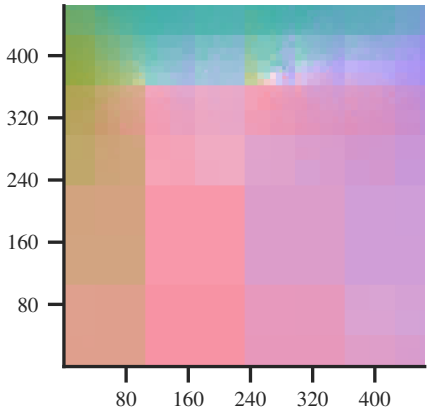


Figure 5.1: The approximate field of a small simulation, as encoded by an octree with vector summaries.

Quadrupole representations improve on this, enabling the field to vary across each node. This first-order approximation is still very inaccurate though, and dramatic discontinuities are still visible in Fig. 5.2.

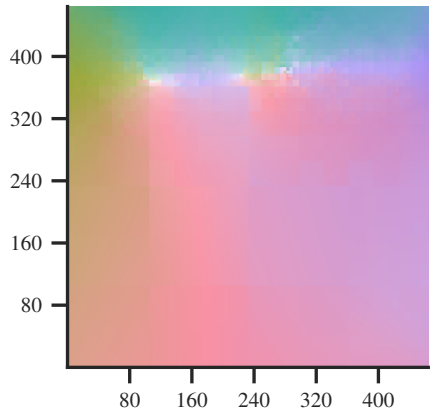
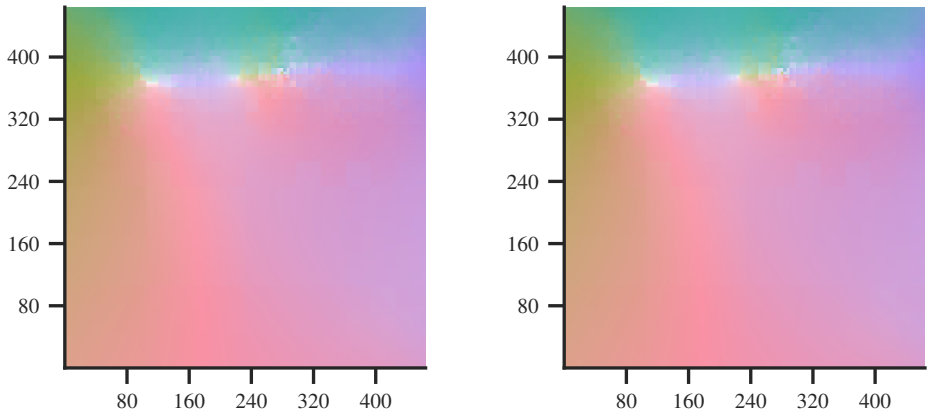


Figure 5.2: The approximate field of a small simulation, as encoded by an octree with quadrupole summaries.

Higher-order representations are progressively more accurate. In Fig. 5.3, the low-density areas already look smooth to the naked eye, despite being made of only a handful of large nodes. The high-order nodes accurately encode the deviation across each node.



(a) Octupole (b) Hexadecupole

Figure 5.3: Approximate fields for octrees with higher order node summaries.

Some discontinuity remains visible in high-density areas. This is because direct interactions are ignored in these plots, and in the near-field these missing interactions dominate the overall field.

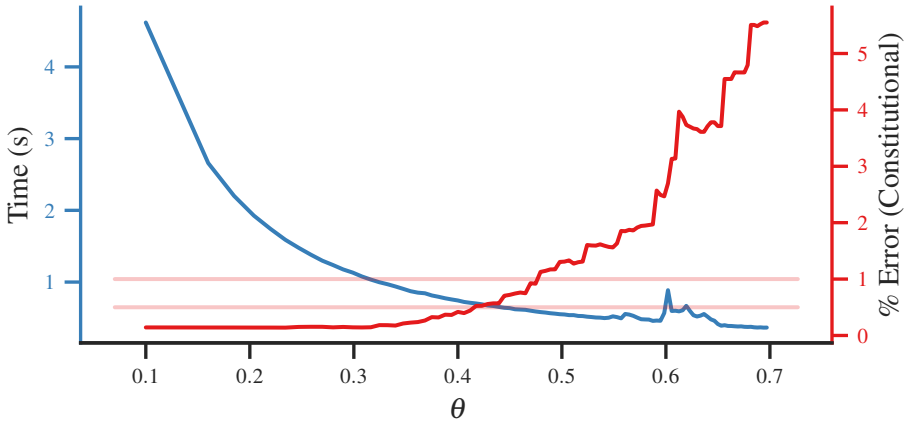
5.1.2 Optimal Multipole Order

Higher multipole orders achieve higher accuracy, but with a performance cost due to their higher memory usage and more complex operations. This higher accuracy

can allow for higher values of θ, increasing the approximation ratio and ultimately improving performance. In order to achieve the best possible performance of each algorithm, we need to select the most appropriate multipole order.

The optimal solution will depend on the desired level of accuracy. In Springel et al. [2021], this is evaluated by plotting error against runtime for a range of θ values, using a dual-axis. Figure 5.4 is an example of this sort of plot, showing both error and iteration time for a solver on the AGORA-LOW dataset. This visualization is information-dense, but it can be difficult to read. We can see that runtime goes down and error rises as θ is increased, but extracting a meaningful evaluation of the solver's performance is more difficult. This requires first finding where the error line meets the desired error, and then finding the time value for that same θ. The two horizontal lines indicate 0.5% and 1.0% error, respectively.

Figure 5.4: Time and Error for a Quadrupole Barnes-Hut solver with varied θ.



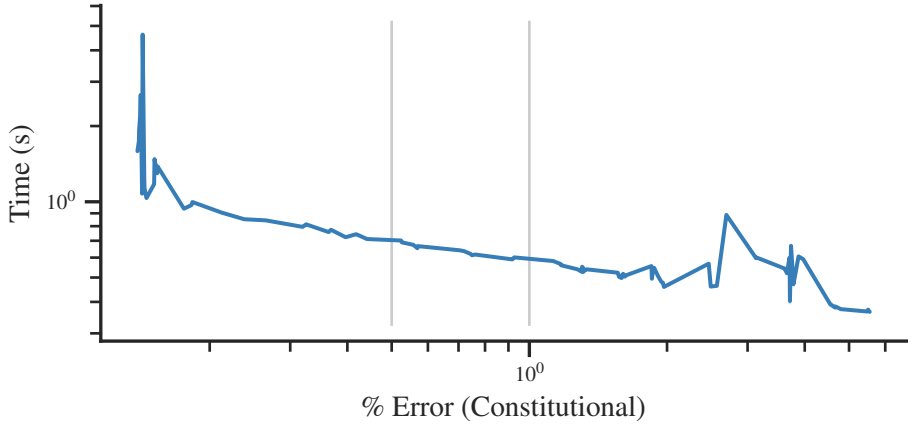
We can improve on this by plotting the data instead as a parametric equation, where $t = \theta$. This more directly shows the tradeoff between accuracy and performance, because the runtime needed to achieve a specific accuracy target is clear. In Fig. 5.5, there is a clear reduction in runtime as we relax our error limit. Here, 0.5% and 1.0% error are indicated by vertical lines.

We can use this technique to compare several multipole orders. Figure 5.6 shows the effect of the multipole order on each solver's runtime-accuracy tradeoff. Note that for dual-tree algorithms, the passive tree always uses multipoles fields that are one order higher than the multipole moments. This means *Quadrupole* refers to a solver with a quadrupole moment and an octupole field, *Hexadecupole* refers to a solver with a hexadecupole moment and a triacontadyupole field.

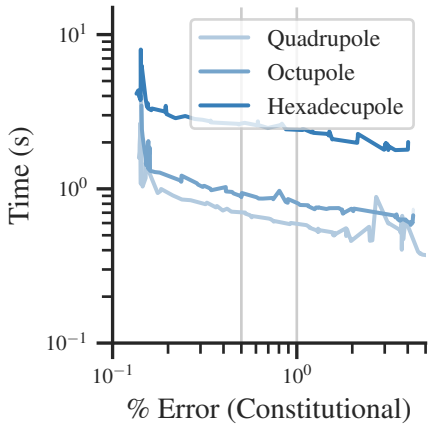
Figures 5.6a and 5.6b show the accuracy curves for the two single-tree algorithms. We can see that the quadrupole variant of both algorithms is optimal for both. This is because the accuracy improvement due to the higher order multipole moment is small. The runtime penalty of the more expensive interactions outweighs the runtime advantage produced by raising θ to achieve the same target accuracy.

Figures 5.6c and 5.6d show that the dual-traversal algorithms benefit more from higher-order multipoles. For both FMM and Dual Hierarchy, Quadrupole and Octupole solvers are competitive across most of the domain. In both cases, the variants

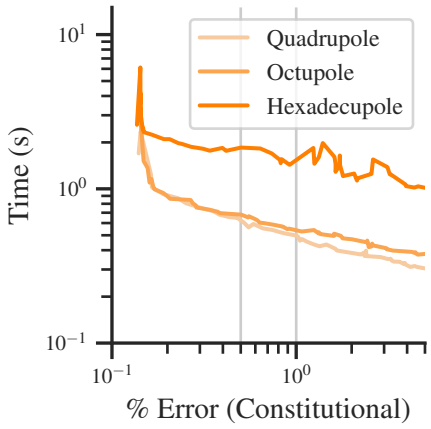
Figure 5.5: Time vs Error for a Quadrupole Barnes-Hut solver with varied θ.



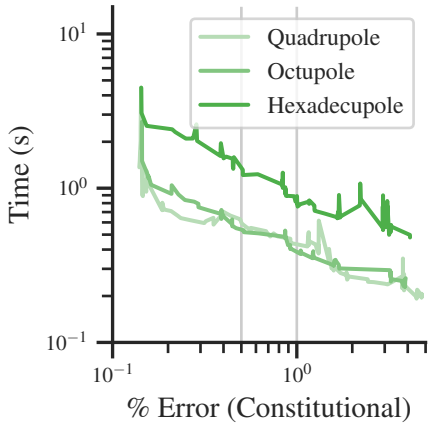
(a) Barnes-Hut



(b) Linear-BVH-Barnes-Hut



(c) FMM



(d) Dual Hierarchy

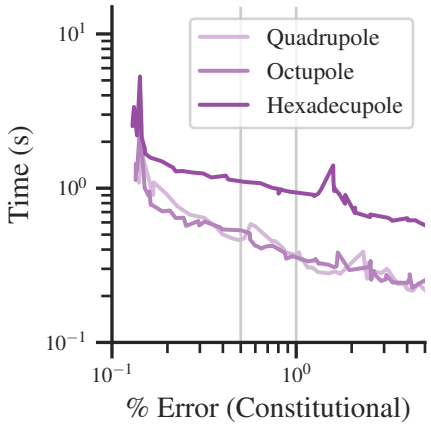


Figure 5.6: Time vs Error for each solver, with each multipole order (AGORA-LOW).

have near-equivalent performance in the 0.5% to 1.0% error range and can be used interchangeably.

In all tests, the Hexadecupole variant of the solver underperforms. This makes sense, as there are diminishing returns to the accuracy benefit of higher orders. The hexadecupole approximation improves accuracy only marginally over the octupole, but hexadecupole operations are nearly twice as complex as the octupole equivalents.

As we increase n, higher-order multipoles become more favorable. For AGORA-MED and AGORA-HI datasets, octupoles become the best choice. With higher n, a higher value of θ is more beneficial, and compute time is more dominated by the particle-particle interactions which could not be eliminated by node approximations. These factors begin to outweigh the compute costs of more elaborate approximations.

For less strict accuracy requirements such as RMS (1.0%), quadrupoles remain optimal at large scales. In the next section, we use quadrupoles when testing with RMS error, and octupoles when testing with constitutional error.

5.2 Complexity

Here, we examine the computational complexity of each algorithm with respect to the number of particles, and how they scale in practice with datasets of increasing size.

Section 5.2.1 breaks down the complexity of the different parts of each algorithm, and how that affects the complexity of the whole. Section 5.2.2 examines the results of real benchmarks of each algorithm.

5.2.1 Theoretical Complexity

The tested solvers can all be separated into three main stages, and the theoretical complexity of these stages can be found in Table 5.1.

The tree construction stage includes all setup required for each iteration before the traversal can begin. For Barnes-Hut and FMM, this means building the octree, which can be done in $O(n\log(n))$ time. For Linear-BVH-Barnes-Hut, this includes generating Morton codes for each particle ($O(n)$), a radix-sort of particles by their Morton codes ($O(n)$), and the construction of bounding boxes to make a tree from the sorted particles ($O(n)$). The Dual Hierarchy algorithm must construct both an octree and a Linear-BVH, with overall complexity of $O(n\log(n))$.

The traversal stage includes the process of finding the acceleration for each particle. In Barnes-Hut and Linear-BVH-Barnes-Hut, this refers to the recursive active traversal including decay to direct interactions, which is done in $O(n\log(n))$ time. In the FMM and Dual Hierarchy algorithms, this refers to the dual traversal and the collapse of the passive tree.

In this implementation, an Octree is constructed in $O(n\log(n))$ time, and a Linear-BVH is constructed in $O(n)$ time, true to its namesake. Active traversal with decay to naive interactions is done in $O(n\log(n))$ time, and dual traversal can be done in $O(n\log(\theta))$ time. Dual traversal originally required a second descent with the same complexity to collapse the passive tree, but this was made unnecessary by the implicit passive tree optimization.

The force application step includes the process used to update velocities and positions, in this case, simple Newtonian integration. This step is generally negligible, even for more complex integrators.

Table 5.1: Theoretical runtime complexity for the stages of each algorithm

Solver	Tree Construction	Traversal	Force Application	Overall
Barnes-Hut	$O(n\log(n))$	$O(n\log(n))$	$O(n)$	$O(n\log(n))$
LBVH Barnes-Hut	$O(n)$	$O(n\log(n))$	$O(n)$	$O(n\log(n))$
FMM	$O(n\log(n))$	$O(n\log(\theta))$	$O(n)$	$O(n\log(n))$
Dual Hierarchy	$O(n\log(n)+n)$	$O(n\log(\theta))$	$O(n)$	$O(n\log(n))$

Memory complexity is expected to be linear for all algorithms, as the memory complexity of both Octree and Linear-BVH is $O(n)$.

5.2.2 Measured Performance

We can measure true performance by timing our implementations on several datasets of different sizes. θ is calibrated to meet target error limits for the AGORA-LOW dataset using the process outlined in Section 4.5.1.

Table 5.2 shows the runtimes of each solver on the AGORA datasets. Each result is the median of 5 runs, each on a different node with the same provisioned resources.

Table 5.2: Median solver runtimes on AGORA datasets, 0.5% constitutional error.

N Solver		112,500	1,125,000	11,250,000
Barnes-Hut	(Quadrupole)	2.42 s	62.55 s	1186.68 s
Barnes-Hut	(Octupole)	2.30 s	54.35 s	902.19 s
Linear-BVH-Barnes-Hut	(Quadrupole)	2.22 s	63.58 s	1276.74 s
Linear-BVH-Barnes-Hut	(Octupole)	2.14 s	57.59 s	1050.04 s
FMM	(Quadrupole)	**1.23 s**	**19.02 s**	275.53 s
FMM	(Octupole)	1.27 s	19.41 s	**267.25 s**
Dual Hierarchy	(Quadrupole)	1.60 s	38.43 s	568.12 s
Dual Hierarchy	(Octupole)	1.38 s	25.94 s	433.06 s

Here we see that the dual traversal algorithms outperform the single traversal algorithms at all scales. The different multipole orders are competitive at small scales, but octupole solvers are generally faster and uniformly outperform their quadrupole counterparts at the largest scales.

Linear-BVH-Barnes-Hut is faster than the traditional octree variant at small scales, but has worse scaling as n increases, making Octupole Barnes-Hut the best single-traversal algorithm. For the largest dataset, it is over 15% faster than Linear-BVH-Barnes-Hut.

The dual-traversal solvers are significantly faster than the single-traversal versions. For the smallest datasets, the dual-traversal algorithms are all more than 25% faster

than the fastest single traversal algorithm. For larger datasets, this gap widens; switching to dual traversal cuts runtime in half or more.

FMM outperforms the adapted Dual Hierarchy algorithm for all datasets, and this advantage only increases with size. For the largest dataset, Octupole FMM is nearly 40% faster than Octupole Dual Hierarchy.

These differences become more exaggerated when we relax the accuracy requirements in Table 5.3. 1.0% RMS error is the least strict error requirement we see used in contemporary survey papers. With looser restrictions on error, higher values of θ can be used. This benefits all the algorithms tested and also means that the quadrupole variants are consistently the fastest. We also see that the advantage that FMM has over Dual Hierarchy grows with increased θ values, and in some cases, it is as much as $3\times$ faster.

Table 5.3: Median solver runtime on AGORA datasets, 1.0% RMS error.

N Solver		112,500	1,125,000	11,250,000
Barnes-Hut	(Quadrupole)	1.27 s	25.36 s	404.91 s
Barnes-Hut	(Octupole)	1.73 s	37.18 s	574.29 s
Linear-BVH-Barnes-Hut	(Quadrupole)	1.28 s	29.50 s	462.25 s
Linear-BVH-Barnes-Hut	(Octupole)	1.55 s	37.18 s	591.29 s
FMM	(Quadrupole)	**0.46 s**	**6.41 s**	**86.49 s**
FMM	(Octupole)	0.79 s	11.27 s	145.62 s
Dual Hierarchy	(Quadrupole)	0.84 s	17.23 s	267.78 s
Dual Hierarchy	(Octupole)	0.87 s	15.85 s	289.79 s

We can take a more granular look at relative performance by timing the solvers on randomly generated datasets in a variety of sizes. These datasets are produced using Perlin noise to avoid some of the issues of uniformly distributed points, but still have far lower dynamic range than the realistic datasets provided by AGORA. As such, magnitudes can be misleading, but they are still useful for making qualitative observations about scaling behavior.

In Figures 5.7 and 5.8, we see results that generally agree with the AGORA results, though the differences between solvers are reduced. There is a clear difference in runtime complexity, with the single traversal solvers showing a characteristic upward curve that matches the theoretical $O(n\log(n))$ complexity, and the dual-traversal algorithms appearing closer to linear, which matches $O(n\log(\theta))$ with fixed θ.

FMM still outperforms Dual Hierarchy, but by a smaller margin than in Table 5.2. Linear-BVH-Barnes-Hut outperforms Barnes-Hut across a wider range of simulation sizes. This may be due to the low dynamic range resulting in more similar tree structures between the two algorithms.

As in Table 5.3, the gap between FMM and Dual Hierarchy widens when we relax our accuracy requirement. Octree-based solvers appear to benefit disproportionately from the higher θ values.

Common observations between the two datasets are that the dual-traversal algorithms handily outperform both single-traversal algorithms, and that FMM reliably

Figure 5.7: Solver times on Perlin noise datasets, 0.5% constitutional error.

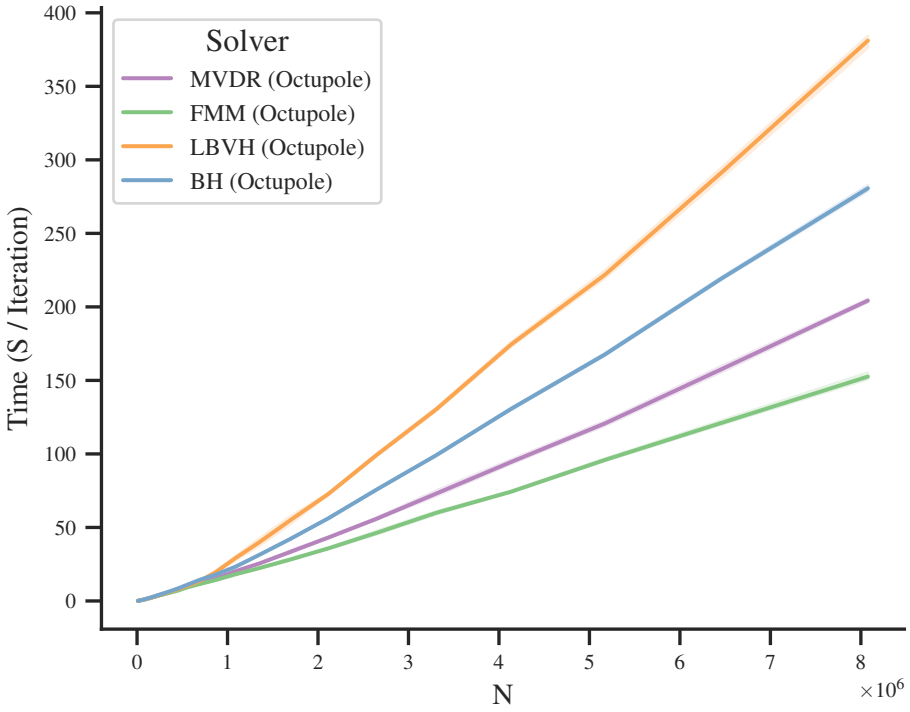
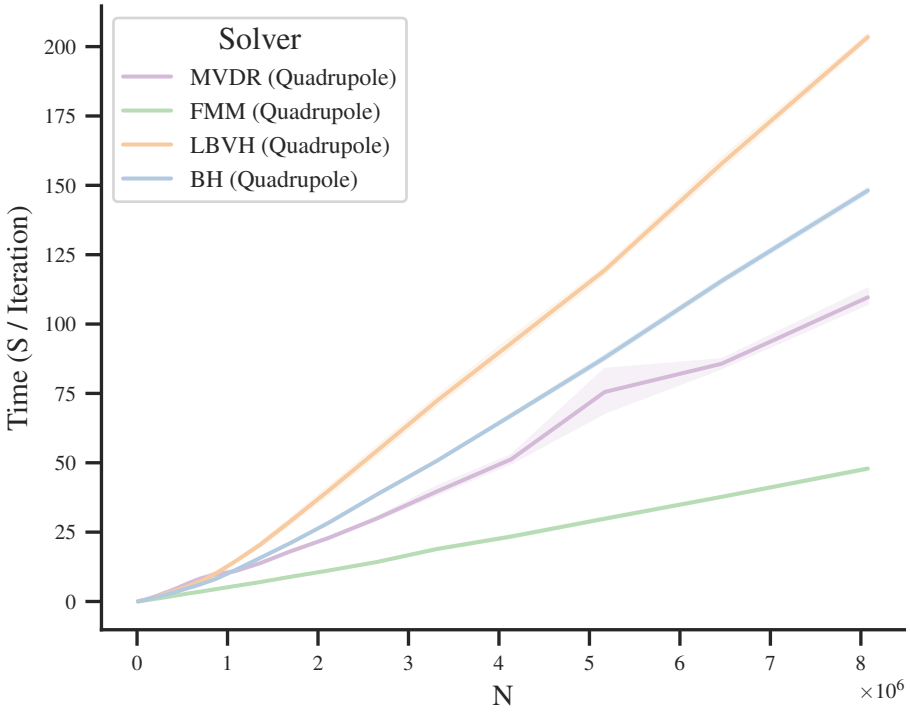


Figure 5.8: Solver times on Perlin noise datasets, 1.0% RMS error.



outperforms Dual Hierarchy, with a lead that grows as n increases. The same patterns continue for benchmarks with uniformly distributed particles. Because these datasets are very different, we speculate that these observations will remain true for a broader range of datasets.

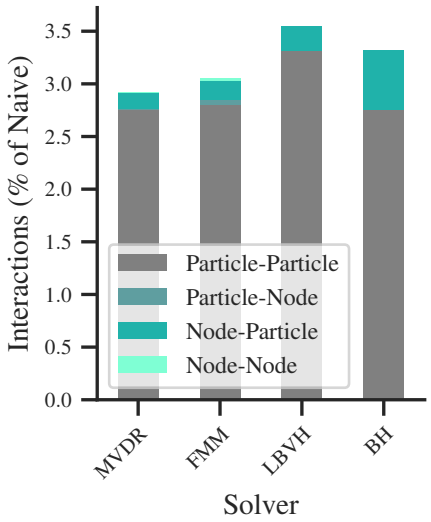
The fact that FMM has the biggest advantage on the lower dynamic-range Perlin noise dataset with the lax accuracy requirement is significant; of the scenarios tested, this is the most similar to t-SNE minimization.

5.3 Approximation Ratio

By counting interactions directly, we can make more fundamental observations about each algorithm and how they compare to one another. This is done using the interaction counting solver discussed in Section 4.3.3. We observe that the Dual Hierarchy algorithm improves on FMM in multiple key metrics; this may indicate that its lackluster performance may not be intrinsic to the algorithm itself.

Figure 5.9 shows the number of interactions computed by each solver, as well as a breakdown of what types of interactions are done. For all solvers, the majority of interactions are computed directly between particles, but this only accounts for a small fraction of the total interactions the naive algorithm would require. The remainder of the naive interactions are replaced by node-node, node-particle, and particle-node interactions.

Figure 5.9: Approximation ratios of each solver (AGORA-LOW), with a breakdown of interaction types.



For the single-tree Linear-BVH and Barnes-Hut algorithms (right), only node-particle approximations are done. These compute the force from active nodes applied to individual passive particles.

The dual-tree algorithms are also dominated by these node-particle approximations. This is explained by our 2:1 ratio between active nodes and passive nodes; the

passive side of the traversal typically reaches a leaf first, so particle-node approximations are unlikely. Table 5.4 shows the exact count of each type of interaction. While these types of interactions are uncommon, they do still occur.

Table 5.4: Interaction counts and approximation ratio for each solver (AGORA-LOW)

Solver	Particle-Particle	Particle-Node	Node-Particle	Node-Node	Approximation Ratio
Barnes-Hut	348,264,716	0	71,044,519	0	0.033131 (30.1×)
Linear-BVH-Barnes-Hut	419,765,992	0	29,114,676	0	0.035467 (28.2×)
FMM	355,320,203	4,938,173	23,062,656	3,051,901	0.030528 (32.7×)
Dual Hierarchy	348,906,940	913,790	19,203,761	214,230	**0.029174** (34.3×)

Unintuitively, the Dual Hierarchy algorithm performs fewer approximate interactions of each sort than FMM, and yet also performs fewer direct interactions. This gives it the best approximation ratio, reducing the number of interactions vs. the naive algorithm by a factor of 34. This is possible because the approximate interactions are done between larger nodes. In dual-hierarchy, 16 node-node interactions can be replaced by a single interaction one level shallower in the tree.

As discussed in Section 4.2.3, not all interactions are equivalent. Direct interactions are simple and can be vectorized to some degree, but interactions involving nodes require multipole math, which can be very expensive.

As a result, the Dual Hierarchy algorithm appears to have a larger advantage when we measure the runtime instruction count directly (Table 5.5). Dual Hierarchy computes approximately 4.4% fewer interactions than FMM, but does so by executing 6.1% fewer instructions. This is likely because fewer of its interactions are between nodes.

A more dramatic result is that the Linear-BVH variant of Barnes-Hut has a lower instruction count despite performing more interactions. This is explained by the fact that it does significantly fewer node-particle interactions than the octree variant; most of its additional interactions are direct and therefore cheaply computed.

Table 5.5: Instruction counts for each solver (AGORA-LOW)

Solver	Instruction Count
Barnes-Hut	75,816,501,410
Linear-BVH-Barnes-Hut	65,252,024,456
FMM	51,128,687,535
Dual Hierarchy	**47,992,604,772**

5.4 Understanding the Performance Disparity

Interaction counts suggest that the adapted Dual Hierarchy algorithm should outperform the state-of-the-art, but that does not match the actual measured performance. In this section, we evaluate several potential explanations for why that might be the case. Section 5.4.1 shows that the time taken to construct the second tree cannot explain the

disparity. Section 5.4.2 examines the nature of the interactions being done, and shows that while the independent trees don't always produce ideal approximations, they are ultimately advantageous. Section 5.4.3 discovers that while the memory usage of the Dual Hierarchy algorithm is relatively small, its access patterns could hinder its performance by reducing the cache hit rate.

5.4.1 Tree Construction

The Dual Hierarchy algorithm must spend additional time constructing its second independent tree, but in practice, this cost is not enough to explain the performance difference.

Figure 5.10: Tree Construction Times on randomly generated data

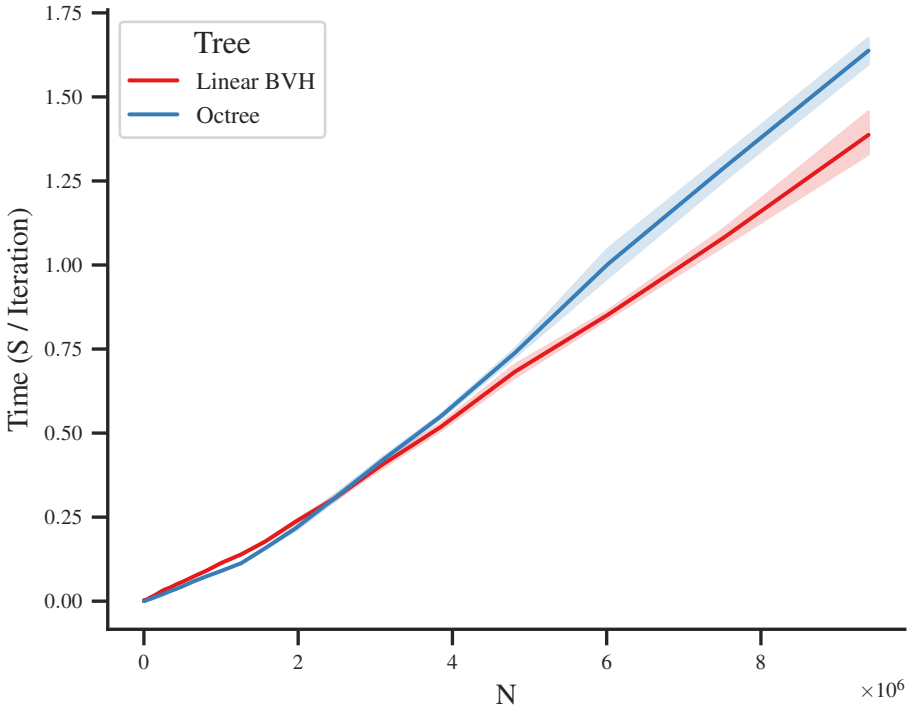
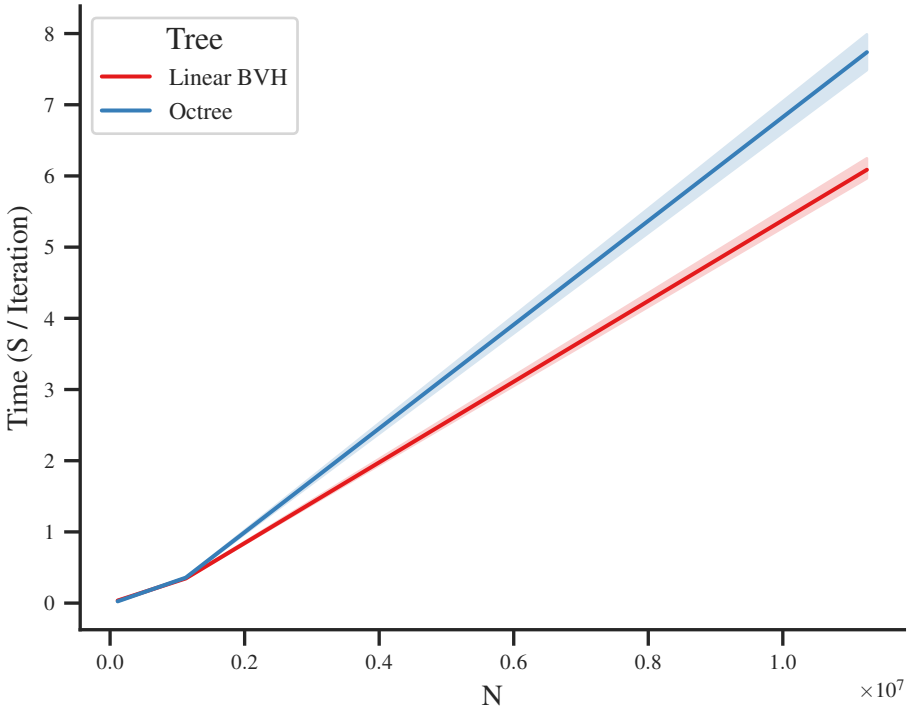


Fig. 5.10 shows that the penalty for the $\log(n)$ component of octree construction is relatively small. Despite this, Linear-BVH construction is faster at large scales. The linear portion of the Linear-BVH construction method has a lower coefficient, and the constant overhead is small. Both construction methods are extremely fast, only requiring more than one second after n reaches 6M particles.

Testing on real data shows that performance suffers with higher dynamic range, as the tree must be less balanced. Despite this, tree construction only accounts for a small percentage of overall runtime. Both tree construction algorithms are parallel, but have some unavoidable sequential components when dividing the first several levels of the tree. The dual-hierarchy algorithm performs the two constructions in parallel with one another, increasing CPU occupancy during this stage. The total construction time for

Figure 5.11: Tree Construction Times on the AGORA dataset



both trees accounts for less than 4% of the overall runtime, which is dominated by traversal.

Both construction algorithms are expressed as sorts: the octree is built by recursive partitioning of the particle list, and the Linear-BVH is built using a Radix sort. This means that for future constructions when the particles have only moved a small amount, the list is already mostly sorted. Stable variants of both sorting algorithms are used, so successive tree constructions are significantly faster. When applying the algorithms to long-running simulations, the tree construction time becomes even less significant.

5.4.2 Node-size Mismatch

The performance disparity may be due in part to the types of interactions allowed by the descent criterion, in combination with the descent algorithm. This is the problem that adaptive-descent attempts to solve. We can determine this using the interaction-tracking introduced in Section 4.3.3.

For reference, Fig. 5.12a is a heatmap that shows a sample of the node-particle interactions performed by the Barnes-Hut algorithm while calculating forces on the AGORA-LOW dataset. The recursive descent algorithm begins with the largest, furthest interactions at the top right of the plot and computes progressively nearer interactions, moving toward the bottom left.

We see that octree node sizes are separated into discrete levels, because the dimensions of the nodes at each level of the tree are uniform. Interactions follow a clear

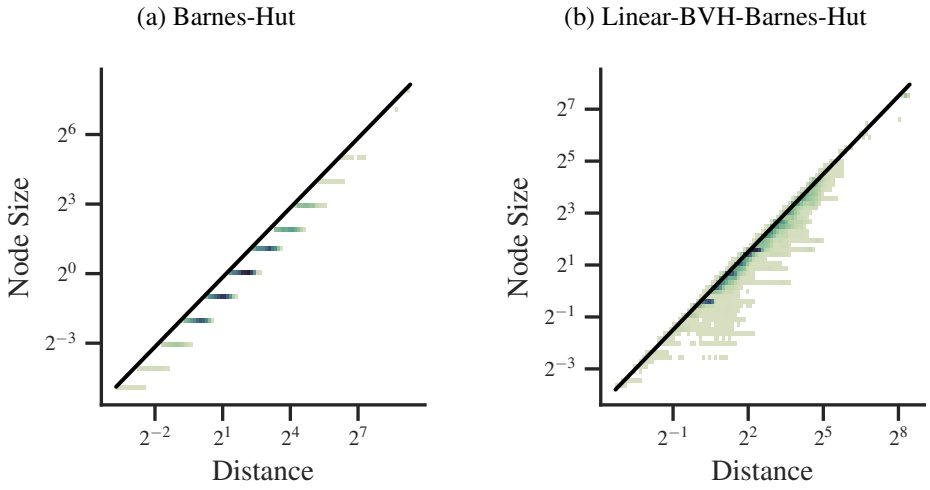


Figure 5.12: Interactions performed by both single-tree algorithms, node size vs distance between nodes.

stair-step pattern because the descent algorithm works to maximize node sizes while keeping the ratio of side length and distance bounded by θ, drawn as a diagonal line.

We also see that most of the interactions are done between mid-sized nodes at medium range. There are fewer large nodes, so fewer interactions between them. The smallest nodes are only present in high-density areas, so they also appear less frequently in our interaction tracking.

The interactions of the Linear-BVH-Barnes-Hut algorithm can be seen in Fig. 5.12b. This plot has a radically different appearance. Because the bounding boxes are tight-fitting, their sizes are not uniform at each level of the tree. The two-way binary splits also add granularity – the Linear-BVH is split into three times as many levels as the octree. There is still a small amount of stair-stepping visible, and this is likely because in high-density regions, Linear-BVH nodes will be closer to fully and uniformly tiling the space, as octree nodes do.

The bulk of interactions are also closer to the line defined by theta. Remember that for our theoretical "optimal" solver discussed in Section 4.4.1, every interaction would be on this line.

When we expand this visualization to dual-traversal algorithms like FMM and Dual Hierarchy, we have a second node to examine. We can use the maximum of the two node sizes to see how close the solvers come to optimal approximation.

Figure 5.13a shows that the interactions of the FMM algorithm are split into discrete levels with a clear stair-stepping pattern, just as with Barnes-Hut. We only include interactions between nodes here, so there are fewer levels with higher nodes. Smaller nodes are more commonly accounted for in particle-node interactions when the algorithm decays to active-tree descent.

Figure 5.13b shows the same behavior as Linear-BVH-Barnes-Hut, with a greater variety of node sizes which are generally closer to the diagonal line indicating *theta*. To understand why this does not convert to greater performance, we also need to look at the ratio between the sizes of the two nodes.

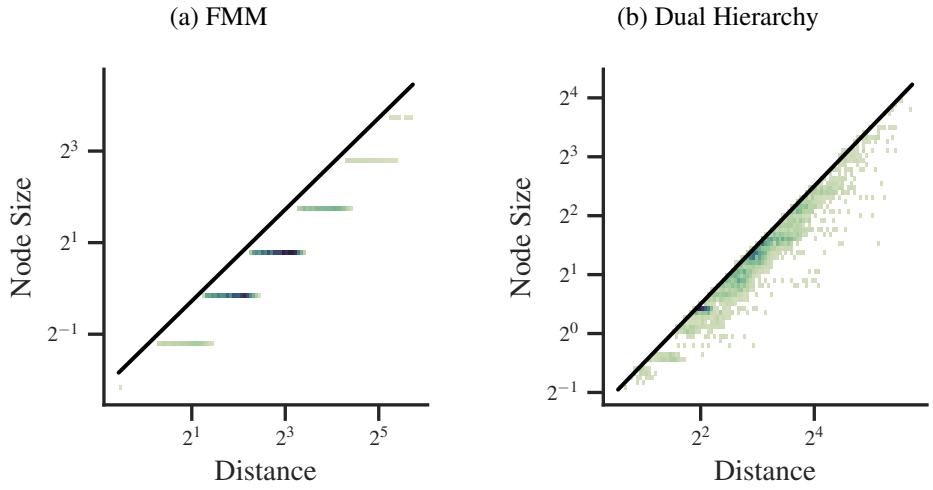


Figure 5.13: Interactions performed by both single-tree algorithms, maximum node size vs distance between nodes.

Fig. 5.14 shows two different views of the same collection of interactions during FMM descent. In Fig. 5.14a we see that the size of the active node is always double that of the passive node. The gray line indicates the intended 2:1 ratio. Figure 5.14b loses size information, but can show trends with respect to distance. Here we see that the ratio between the node sizes is always 2:1, regardless of the distance between the nodes.

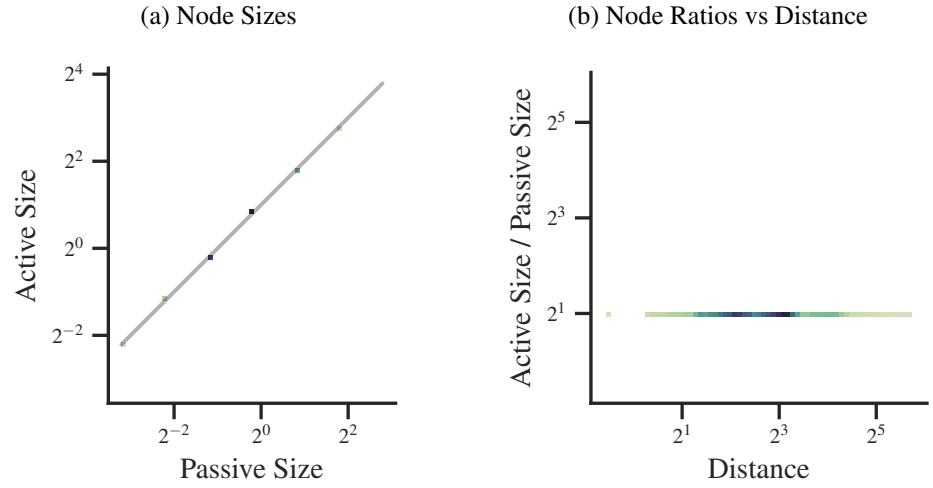


Figure 5.14: Interaction ratios of Node-Node interactions, for FMM.

Figure 5.15 tells a very different story. In Fig. 5.15a, we see that the passive Octree is still split into distinct levels, but the 2:1 ratio is not constant. In some cases, the active node is smaller than optimal, but in most cases, it is larger. This is the predicted issue that motivated the adaptive-descent method explored in Section 4.4.1.

Figure 5.15b shows that the disparity in node sizes has little correlation with distance. This indicates that the difference in sizes comes primarily from the randomness of Linear-BVH sizes. If the disparity came from the greater splitting factor of the octree, we would expect a strong correlation, as the $8\times$ splits of the octree would quickly reach smaller nodes than the $2\times$ splits of the Linear-BVH.

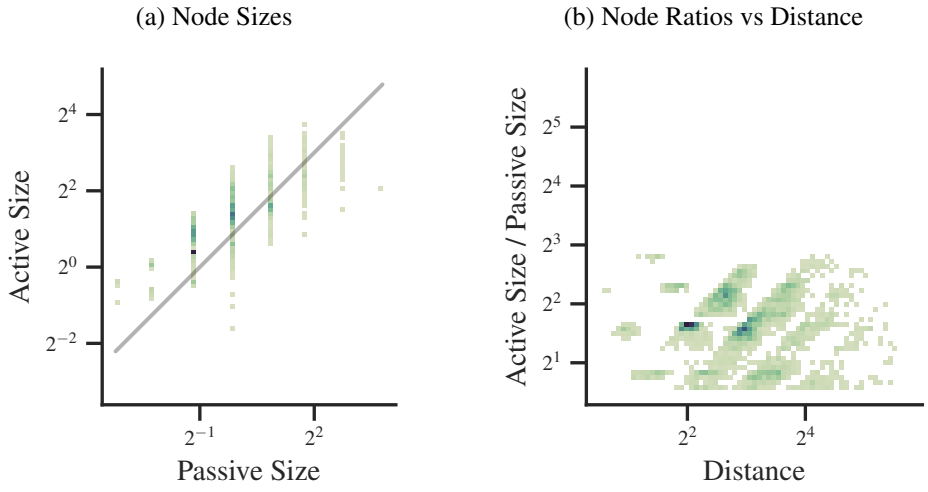


Figure 5.15: Interaction ratios of Node-Node interactions, for Dual Hierarchy.

Table 5.4 demonstrates that the nearer-to-θ interactions allowed by more granular node sizes produce fewer, larger interactions closer to the root of the tree. This advantage outweighs any disadvantage that comes from less-than-optimal node ratios, ultimately allowing the dual hierarchy algorithm to achieve the highest approximation ratio while meeting the same accuracy requirements as the other algorithms. This issue cannot explain the lower-than-expected performance of the new algorithm.

5.4.3 Memory Usage and Access Patterns

The second hierarchy of the Dual Hierarchy algorithm needs to be stored and retrieved, so it stands to reason that the algorithm comes with a memory and cache disadvantage. This may contribute to its failure to outperform FMM.

We can estimate the memory cost of our trees by counting the nodes and multiplying by the size of individual nodes. We may expect Linear-BVH nodes to be much more space-consuming because they need to store their own bounding boxes, but alignment means that this isn't necessarily the case. In Table 5.6, we can see that while the Linear-BVH nodes do require more space to store, this is generally dominated by the costs of storing the multipoles. In the case of an implicit node, which stores no summary, there is no difference in size at all due to the padding added by the compiler.

As seen in Fig. 5.16, This small difference is generally outweighed by the fact that an Linear-BVH more optimally divides its contents. An octree can have many empty or nearly-empty leaf nodes because the arrangement of the nodes is fixed, but the Linear-BVH construction process ensures a more balanced tree.

We should expect the memory use of the trees to grow linearly with n, and this is supported by the data. The AGORA datasets provide the most realistic estimate of

Table 5.6: Node Sizes for each tree type (bytes)

Summary Type	Octree	Linear-BVH
None	48	48
Vector	56	64
Quadrupole	96	104
Octupole	136	144
Hexadecupole	192	200

(a) AGORA Data (b) Random Data

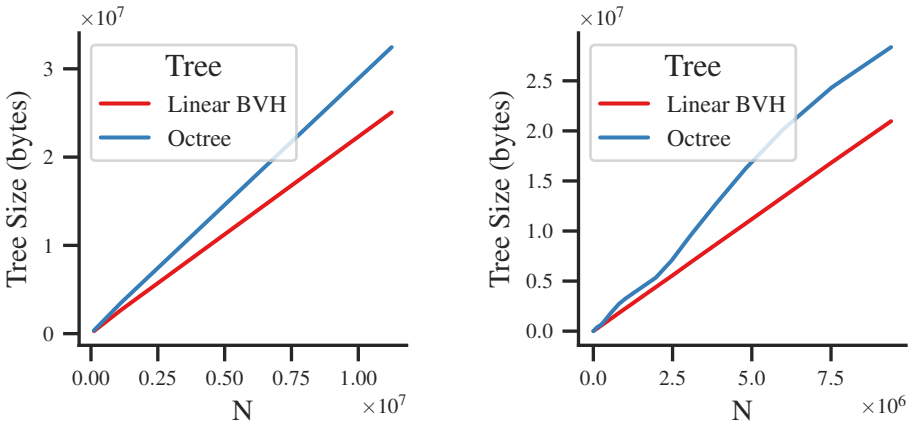


Figure 5.16: The sizes of trees constructed from Perlin noise random data and from AGORA datasets.

memory use, and we can see that the Linear-BVH requires less space to store than the Octree.

We can produce a more granular look at the growth rate using random data. We see that the octree grows in "waves", this is likely because as density increases, each additional level produces an 8-fold increase in nodes, most of which are empty. The next level is not necessary until those new nodes begin to saturate. The Linear-BVH has a much smoother growth curve, because each additional particle is allocated into the tree optimally, only adding new nodes if needed.

Because each leaf can hold many particles, and the tree grows linearly, the memory requirements of either tree are small. The Dual Hierarchy algorithm uses less than 100 megabytes to store both trees on the very largest datasets tested.

Cache Miss Rate

Memory use is not the primary cost of the second tree, however. The dual-tree traversal has less-than-optimal memory access patterns, and this is visible in its cache miss rate.

We can measure the miss rate using the Cachegrind tool, which is a part of the Valgrind memory debugger and profiler. This tool simulates a simplified cache hierarchy, with independent first-level instruction and data caches (I1 and L1, respectively),

and a combined last-level cache (LL). The intermediate cache levels of real modern CPUs are not included in the simulation, and LL is the largest cache level before main memory.

Table 5.7 shows the results for several solvers running on the AGORA-LOW dataset. The solvers have uniformly low instruction miss rates, which makes sense given the limited divergence of the algorithms. They also have last-level miss rates low enough to round down to 0%. This is likely a result of the relatively compact tree representations.

Table 5.7: Simulated data cache miss rates for each solver (AGORA-LOW)

Solver	I1 Miss Rate	L1 Miss Rate	LL Miss Rate
Barnes-Hut (Quadrupole)	0.01%	4.8%	0.0%
Linear-BVH-Barnes-Hut (Quadrupole)	0.01%	7.6%	0.0%
FMM (Quadrupole)	0.01%	**3.1%**	0.0%
Dual Hierarchy (Quadrupole)	0.01%	5.9%	0.0%

The L1 statistics are more informative. Our tree implementations place the children of each node adjacent in memory, and the access pattern of a traversal can make a significant difference in miss rates. Descending the octree for Barnes-Hut means retrieving 8 nodes at a time, but descending the binary tree of the Linear-BVH variant only retrieves 2 nodes at a time. This means that the original Barnes-Hut requires significantly less pointer-chasing. This corresponds with a significant difference in miss rate, as fewer child nodes are already in the cache when needed by the Linear-BVH descent.

The more complex dual-traversal required by FMM actually decreases the miss rate vs the single-traversal Barnes-Hut. This may be because two different traversals are being done over the same tree, so nodes are pulled into the cache for one side of the traversal before they are needed by the other side. Specifically, our 2:1 ratio means that the passive node is always one level deeper than the active. When a node is retrieved to use on the active side of an interaction, is likely already in the cache because it was used on the passive side of an earlier interaction.

The Dual Hierarchy algorithm receives no such benefit, as the two sides of its traversal are pulling from different trees. Because traversal of the active Linear-BVH is done as part of the inner loop of the implicit dual traversal, its nodes are descended in groups. This may improve its cache hit rate, explaining how the miss rate is closer to that of Barnes-Hut than that of Linear-BVH-Barnes-Hut.

These cache misses are likely the main cause of the disparity between instruction count (Table 5.5) and real-life performance (Table 5.2). This also explains why increasing θ widens the gap between FMM and Dual Hierarchy (Table 5.3). As θ increases, more time is spent on traversal and node interactions, and less is spent on direct particle-particle interactions. These direct interactions have similar memory access patterns between all solvers, and Dual Hierarchy is the best at eliminating them because of its high approximation ratio. When the number of direct interactions is reduced across the board, this advantage of Dual Hierarchy becomes less impactful.

To some degree, the poor memory access patterns are intrinsic to the dual hierarchy concept, but it may be possible to mitigate the issue with a better memory layout.

Specifically, the tree could be implemented in tabular format, rather than using pointers and allowing the compiler to place nodes arbitrarily in heap memory. This could be further improved by constructing the active tree breadth-first instead of depth-first, so that nodes of the same depth are adjacent in memory. This would more closely match the order in which they are traversed, potentially improving the usefulness of hardware block prefetching. This could be further supplemented with explicit prefetching in software.

Chapter 6

Conclusion

We have demonstrated that the Dual Hierarchy algorithm can be adapted to gravitational n-body. We have then optimized the adapted algorithm for use on the CPU with significantly reduced memory requirements. While our implementation of the new algorithm performs significantly better than multipole Barnes-Hut, it is ultimately slower than an equivalent FMM implementation.

Detailed tracking of the adapted algorithm indicated that on a sample problem, it was able to achieve the same accuracy as FMM while computing 4.4% fewer gravitational interactions. This is made possible by the more granular range of sizes in the tight-fitting nodes of its Linear-BVH. Because a greater portion of these interactions are directly between particles rather than the more expensive node interactions, this translates to a 6.1% reduction in runtime instruction count.

Ultimately, this is not converted to a proportional reduction in runtime. Dual Hierarchy consistently outperforms traditional single-traversal algorithms, providing over a 51% runtime reduction vs. Barnes-Hut on the largest datasets. The state-of-the-art FMM has even better performance on these datasets, with a total runtime as much as 38% lower than Dual Hierarchy while meeting the same accuracy requirements. FMM also provides a speedup for the scenarios most similar to t-SNE minimization, with a 45% runtime reduction on the largest Perlin noise datasets when accuracy requirements are relaxed.

After ruling out several other explanations, this discrepancy in algorithmic performance and measured runtime is attributed to the better memory access patterns of FMM. This is supported by the cache miss rates of the algorithms on simulated hardware. Because only one tree is being traversed instead of two, the hardware prefetcher can be more effective, increasing the hit rate.

Because of this advantage, we recommend the exploration of FMM for t-SNE. Dual-traversal on an Octree was originally dismissed for use on t-SNE by the original Van Der Maaten [2014] paper, due to poor accuracy and performance. Because of the similarity of the t-SNE and gravity equations, a multipole expansion would be trivial, and our experiments show that the use of multipoles is critical for FMM to produce accurate results. We also note that the 2014 algorithm used symmetric interactions and did not retain lists of the particles contained by each node, both of which may have been factors in its poor performance. From the success of Dual Hierarchy for t-SNE, we now know that an efficient GPU implementation of dual traversal is possible, so adapting FMM is a natural next step.

Bibliography

Sverre J. Aarseth. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, Cambridge, 2003. ISBN 9780521432726. doi: 10.1017/CBO9780511535246. URL https://www.cambridge.org/core/books/gravitational-nbody-simulations/A5D1D86EA634C9D354B7C82C029D6933.

Srinivas Aluru. Greengards N-Body Algorithm is not Order N. *SIAM Journal on Scientific Computing*, 17(3):773–776, May 1996. ISSN 1064-8275, 1095-7197. doi: 10.1137/S1064827593272031. URL http://epubs.siam.org/doi/10.1137/S1064827593272031.

John Ambrosiano, Leslie Greengard, and Vladimir Rokhlin. The fast multipole method for gridless particle simulation. *Computer Physics Communications*, 48 (1):117–125, January 1988. ISSN 0010-4655. doi: 10.1016/0010-4655(88)90029-X. URL https://www.sciencedirect.com/science/article/pii/001046558890029X.

Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct Trees in Practice. pages 84–97, Philadelphia, PA, January 2010. Society for Industrial and Applied Mathematics. ISBN 9780898719314 9781611972900. doi: 10.1137/1.9781611972900.9. URL http://epubs.siam.org/doi/abs/10.1137/1.9781611972900.9.

J. S. Bagla. TreePM: A code for Cosmological N-Body Simulations. *Journal of Astrophysics and Astronomy*, 23(3-4):185–196, December 2002. ISSN 0250-6335, 0973-7758. doi: 10.1007/BF02702282. URL http://arxiv.org/abs/astro-ph/9911025. arXiv:astro-ph/9911025.

Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986. ISSN 0028-0836, 1476-4687. doi: 10.1038/324446a0. URL http://www.nature.com/articles/324446a0.

Dino Boccaletti and giuseppe Pucacco. Chaos in N-body systems. *Planetary and Space Science*, 46(11):1557–1566, November 1998. ISSN 0032-0633. doi: 10.1016/S0032-0633(97)00214-6. URL https://www.sciencedirect.com/science/article/pii/S0032063397002146.

David M. Chan, Roshan Rao, Forrest Huang, and John F. Canny. T-SNE-CUDA: GPU-Accelerated T-SNE and its Applications to Modern Data. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 330–338, September 2018. doi: 10.1109/CAHPC.2018.8645912. ISSN: 1550-6533.

Walter Dehnen. A Very Fast and Momentum-Conserving Tree Code. *The Astrophysical Journal*, 536(1):L39–L42, June 2000. ISSN 0004637X. doi: 10.1086/312724. URL http://arxiv.org/abs/astro-ph/0003209. arXiv:astro-ph/0003209.

Marios D. Dikaiakos and Joachim Stadel. A performance study of cosmological simulations on message-passing and shared-memory multiprocessors. In *Proceedings of the 10th international conference on Supercomputing - ICS '96*, pages 94–101, Philadelphia, Pennsylvania, United States, 1996. ACM Press. ISBN 9780897918039. doi: 10.1145/237578.237590. URL http://portal.acm.org/citation.cfm?doid=237578.237590.

J. W. Eastwood, R. W. Hockney, and D. N. Lawrence. P3M3DPThe three-dimensional periodic particle-particle/ particle-mesh program. *Computer Physics Communications*, 19(2):215–261, April 1980. ISSN 0010-4655. doi: 10.1016/0010-4655(80)90052-1. URL https://www.sciencedirect.com/science/article/pii/0010465580900521.

Cong Fu, Yonghui Zhang, Deng Cai, and Xiang Ren. AtSNE: Efficient and Robust Visualization on GPU through Hierarchical Optimization. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pages 176–186, New York, NY, USA, July 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3330834. URL https://doi.org/10.1145/3292500.3330834.

L Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, December 1987. ISSN 0021-9991. doi: 10.1016/0021-9991(87)90140-9. URL https://www.sciencedirect.com/science/article/pii/0021999187901409.

Tomoaki Ishiyama, Kohji Yoshikawa, and Ataru Tanikawa. High Performance Gravitational N-body Simulations on Supercomputer Fugaku. In *International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2022, pages 10–17, New York, NY, USA, January 2022. Association for Computing Machinery. ISBN 9781450384988. doi: 10.1145/3492805.3492816. URL https://doi.org/10.1145/3492805.3492816.

Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas Quinn. Massively parallel cosmological simulations with ChaNGa. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, April 2008. doi: 10.1109/IPDPS.2008.4536319. ISSN: 1530-2075.

Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling Hierarchical N-body Simulations on GPU Clusters. In

SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11, November 2010. doi: 10.1109/SC.2010.49. ISSN: 2167-4337.

Ji-hoon Kim, Oscar Agertz, Romain Teyssier, Michael J. Butler, Daniel Ceverino, Jun-Hwan Choi, Robert Feldmann, Ben W. Keller, Alessandro Lupi, Thomas Quinn, Yves Revaz, Spencer Wallace, Nickolay Y. Gnedin, Samuel N. Leitner, Sijing Shen, Britton D. Smith, Robert Thompson, Matthew J. Turk, Tom Abel, Kenza S. Arraki, Samantha M. Benincasa, Sukanya Chakrabarti, Colin DeGraf, Avishai Dekel, Nathan J. Goldbaum, Philip F. Hopkins, Cameron B. Hummels, Anatoly Klypin, Hui Li, Piero Madau, Nir Mandelker, Lucio Mayer, Kentaro Nagamine, Sarah Nickerson, Brian W. O'Shea, Joel R. Primack, Santi Roca-Fàbrega, Vadim Semenov, Ikkoh Shimizu, Christine M. Simpson, Keita Todoroki, James W. Wadsley, and John H. Wise. The AGORA High-Resolution Galaxy Simulations Comparison Project. II: Isolated Disk Test. *The Astrophysical Journal*, 833(2):202, December 2016. ISSN 1538-4357. doi: 10.3847/1538-4357/833/2/202. URL `http://arxiv.org/abs/1610.03066`. arXiv:1610.03066 [astro-ph].

George Lake, Neal Katz, Thomas Quinn, and Stadel Joachim. Cosmological N-Body Simulation. In *Proceedings for the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, page 312, February 1995.

George C. Linderman, Manas Rachh, Jeremy G. Hoskins, Stefan Steinerberger, and Yuval Kluger. Efficient Algorithms for t-distributed Stochastic Neighborhood Embedding. *Nature Methods*, 16(3):243–245, March 2019. ISSN 1548-7091, 1548-7105. doi: 10.1038/s41592-018-0308-4. URL `http://arxiv.org/abs/1712.09005`. arXiv:1712.09005 [cs, stat].

Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008. ISSN 1533-7928. URL `http://jmlr.org/papers/v9/vandermaaten08a.html`.

Bruno Henrique Meyer, Aurora Trinidad Ramirez Pozo, and Wagner M. Nunan Zola. Improving Barnes-Hut t-SNE Scalability in GPU with Efficient Memory Access Strategies. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2020. doi: 10.1109/IJCNN48605.2020.9206962. ISSN: 2161-4407.

Bruno Henrique Meyer, Aurora Trinidad Ramirez Pozo, and Wagner M. Nunan Zola. Improving Barnes-Hut t-SNE Algorithmăin Modern GPU Architectures with Random Forest KNN and Simulated Wide-Warp. *ACM Journal on Emerging Technologies in Computing Systems*, 17(4):53:1–53:26, June 2021. ISSN 1550-4832. doi: 10.1145/3447779. URL `https://doi.org/10.1145/3447779`.

Bruno Henrique Meyer, Aurora Trinidad Ramirez Pozo, and Wagner M. Nunan Zola. Global and local structure preserving GPU t-SNE methods for large-scale applications. *Expert Systems with Applications*, 201:116918, September 2022. ISSN 0957-4174. doi: 10.1016/j.eswa.2022.116918. URL `https://www.sciencedirect.com/science/article/pii/S0957417422003530`.

Nicola Pezzotti, Julian Thijssen, Alexander Mordvintsev, Thomas Höllt, Baldur Van Lew, Boudewijn P.F. Lelieveldt, Elmar Eisemann, and Anna Vilanova. GPGPU Linear Complexity t-SNE Optimization. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1172–1181, January 2020. ISSN 1941-0506. doi: 10.1109/TVCG.2019.2934307.

Douglas Potter, Joachim Stadel, and Romain Teyssier. PKDGRAV3: Beyond Trillion Particle Cosmological Simulations for the Next Era of Galaxy Surveys, September 2016. URL http://arxiv.org/abs/1609.08621. arXiv:1609.08621 [astro-ph].

H. Rein and S. F. Liu. REBOUND: an open-source multi-purpose N-body code for collisional dynamics. *Astronomy and Astrophysics*, 537:A128, January 2012. ISSN 0004-6361. doi: 10.1051/0004-6361/201118085. URL https://ui.adsabs.harvard.edu/abs/2012A&A...537A.128R. ADS Bibcode: 2012A&A...537A.128R.

John K. Salmon and Michael S. Warren. Skeletons from the Treecode Closet. *Journal of Computational Physics*, 111:136–155, March 1994. ISSN 0021-9991. doi: 10.1006/jcph.1994.1050. URL https://ui.adsabs.harvard.edu/abs/1994JCoPh.111..136S. ADS Bibcode: 1994JCoPh.111..136S.

Aurel Schneider, Romain Teyssier, Doug Potter, Joachim Stadel, Julian Onions, Darren S. Reed, Robert E. Smith, Volker Springel, Frazer R. Pearce, and Roman Scoccimarro. Matter power spectrum and the challenge of percent accuracy. *Journal of Cosmology and Astroparticle Physics*, 2016(04):047–047, April 2016. ISSN 1475-7516. doi: 10.1088/1475-7516/2016/04/047. URL http://arxiv.org/abs/1503.05920. arXiv:1503.05920 [astro-ph].

Volker Springel, Rüdiger Pakmor, Oliver Zier, and Martin Reinecke. Simulating cosmic structure formation with the GADGET-4 code. *Monthly Notices of the Royal Astronomical Society*, 506(2):2871–2949, July 2021. ISSN 0035-8711, 1365-2966. doi: 10.1093/mnras/stab1855. URL http://arxiv.org/abs/2010.03567. arXiv:2010.03567 [astro-ph].

Mark van de Ruit, Markus Billeter, and Elmar Eisemann. An Efficient Dual-Hierarchy t-SNE Minimization. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):614–622, January 2022. ISSN 1941-0506. doi: 10.1109/TVCG.2021.3114817.

Laurens Van Der Maaten. Accelerating t-SNE using tree-based algorithms. *The Journal of Machine Learning Research*, 15(1):3221–3245, January 2014. ISSN 1532-4435.

Appendix A

Multipole Equations

Given the equation for the force of gravity:

$$F_g = \frac{Gm_1m_2}{\|\Delta\|^2} \tag{A.1}$$

Where Δ is the $\langle \Delta_x, \Delta_y, \Delta_z \rangle$ vector connecting the position of the force-emitting particle to the force receiving particle, with masses m_1 and m_2 respectively, and $\|\Delta\|$ is the magnitude of that vector.

We can produce a series of derivatives $F_g^{(n)}$ of the gravitational field with respect to changes in Δ, from Springel et al. [2021]:

$$(F_g^{(1)})_i = \frac{G}{\|\Delta\|}\Delta_i \tag{A.2}$$

$$(F_g^{(2)})_{ij} = \frac{G}{\|\Delta\|}\delta_{ij} + \frac{G}{\|\Delta\|^2}\Delta_i\Delta_j \tag{A.3}$$

$$\begin{aligned}(F_g^{(3)})_{ijk} = {} & \frac{G}{\|\Delta\|^2}\left(\delta_{ij}\Delta_k + \delta_{jk}\Delta_i + \delta_{ik}\Delta_j\right) + \\ & \frac{G}{\|\Delta\|^3}\Delta_i\Delta_j\Delta_k\end{aligned} \tag{A.4}$$

$$\begin{aligned}(F_g^{(4)})_{ijkl} = {} & \frac{G}{\|\Delta\|^2}\left(\delta_{ij}\delta_{kl} + \delta_{jk}\delta_{il} + \delta_{ik}\delta_{jl}\right) + \\ & \frac{G}{\|\Delta\|^3}\left(\delta_{ij}\Delta_k\Delta_l + \delta_{jk}\Delta_i\Delta_l + \delta_{ik}\Delta_j\Delta_l + \right. \\ & \left.\quad \delta_{il}\Delta_j\Delta_k + \delta_{jl}\Delta_i\Delta_k + \delta_{kl}\Delta_i\Delta_j\right) + \\ & \frac{G}{\|\Delta\|^4}\Delta_i\Delta_j\Delta_k\Delta_l,\end{aligned} \tag{A.5}$$

$$\begin{aligned}(F_g^{(5)})_{ijklm} = {} & \frac{G}{\|\Delta\|^3}\left[\Delta_m\left(\delta_{ij}\delta_{kl} + \delta_{jk}\delta_{il} + \delta_{ik}\delta_{jl}\right) + \right. \\ & \quad \Delta_l\left(\delta_{ij}\delta_{km} + \delta_{jk}\delta_{im} + \delta_{ik}\delta_{jm}\right) + \\ & \quad \Delta_k\left(\delta_{ij}\delta_{lm} + \delta_{jm}\delta_{il} + \delta_{im}\delta_{jl}\right) + \\ & \quad \Delta_j\left(\delta_{im}\delta_{kl} + \delta_{km}\delta_{il} + \delta_{ik}\delta_{lm}\right) + \\ & \quad \left.\Delta_i\left(\delta_{jm}\delta_{kl} + \delta_{jk}\delta_{lm} + \delta_{km}\delta_{jl}\right)\right] +\end{aligned}$$

$$\begin{aligned}
&\frac{G}{\|\Delta\|^4}\left(\delta_{ij}\Delta_k\Delta_l\Delta_m+\delta_{jk}\Delta_i\Delta_l\Delta_m+\delta_{ik}\Delta_j\Delta_l\Delta_m+\right.\\
&\quad\delta_{il}\Delta_j\Delta_k\Delta_m+\delta_{jl}\Delta_i\Delta_k\Delta_m+\delta_{kl}\Delta_i\Delta_j\Delta_m+\\
&\quad\delta_{im}\Delta_j\Delta_k\Delta_l+\delta_{jm}\Delta_i\Delta_k\Delta_l+\delta_{km}\Delta_i\Delta_j\Delta_l+\\
&\quad\left.\delta_{lm}\Delta_i\Delta_j\Delta_k\right)+\\
&\frac{G}{\|\Delta\|^5}\Delta_i\Delta_j\Delta_k\Delta_l\Delta_m, \qquad \text{(A.6)}
\end{aligned}$$

Where Δ is the radial vector connecting centers of the two multipoles and δ is the Kronecker delta.

Appendix B

Exclusion Regions

We need to find the optimal exclusion region for a non-square node, which provides all the same guarantees as the original exclusion region in Fig. 4.5 while having the minimum extents possible, to maximize allowable approximations.

A straightforward first approach is to produce a square exclusion region with a side length double that of the longest side of the node, as shown in Fig. B.1. This clearly provides all of the same guarantees the original exclusion region does, but it may be possible to meet the same accuracy with a smaller region. The exclusion region should be only as large as necessary, because otherwise opportunities for approximation will be unnecessarily discarded.

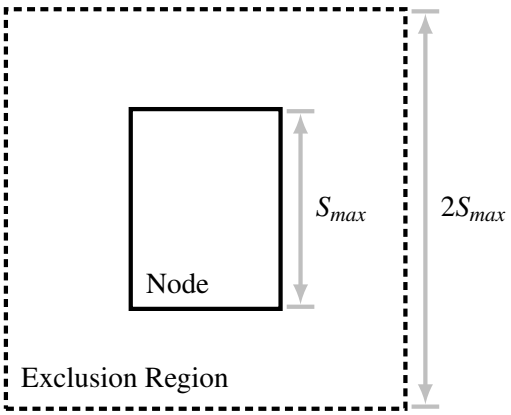


Figure B.1: A rectangular Linear-BVH node with an exclusion region produced from its longest side.

An obvious alternative is to produce the exclusion region by simply dilating the bounding box, as shown in Fig. B.2. This more tightly fits the bounding box, but it may not provide the necessary guarantees. The true goal of the exclusion region is to bound the *Angular Size* of the node with respect to the sampling point. In the original version, an exclusion region with a side length double that of the node ensures that the angular size of the node is never greater than 90°. If we want to use a rectangular bounding box, it should always maintain a buffer which is one half of the side length. The dilation approach is wasteful in this regard, because it has larger bounds on the short sides than on the long ones. It guarantees the same requirement, but may exclude many interactions unecessarily.

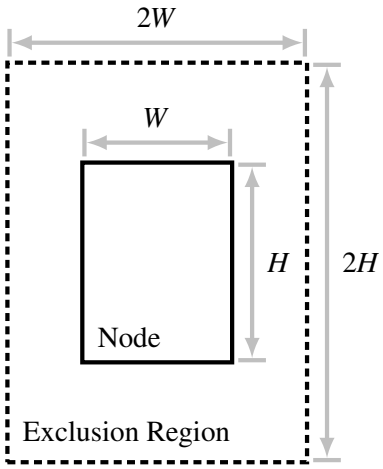


Figure B.2: A rectangular Linear-BVH node with an exclusion region produced by dilating its bounding box.

A rectangular region which guarantees our angular size requirement is shown in Fig. B.3. This is a tighter exclusion region than both previous metrics, and can be considered the optimal rectangle for this purpose. A shape with curved edges which is even tighter may be possible, but it would also making intersection checking more expensive.

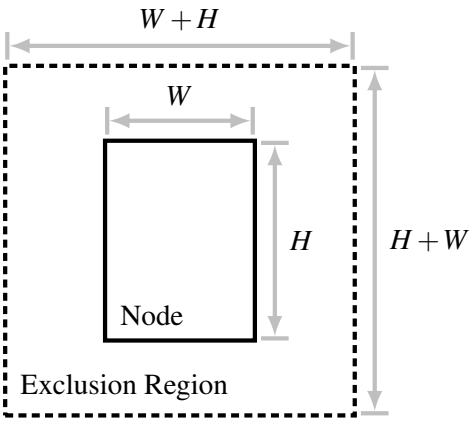


Figure B.3: A rectangular Linear-BVH node and a square exclusion region which fits it optimally.