Contents

Notes on N-Body Simulation **1**
The N-Body Problem . . . 1
Direct Summation . . . 1
Boundaries and Singularities . . . 2
Periodic Boundary Conditions . . . 2
Gravitational Softening . . . 2
The Integrator . . . 3
The Euler Method . . . 3
Velocity Verlet . . . 3
Symplectic Integration . . . 3
The Kick-Drift-Kick Integrator . . . 4
The Particle-Mesh Method . . . 4
Step 1. Finding the Potential . . . 5
Step 2. From Potential to Force . . . 7
Advanced Interpolation . . . 7
The Flaws of Nearest Grid Point (NGP) . . . 7
Cloud-in-Cell (CIC) . . . 8
Implementation: "Splatting" Mass and "Gathering" Forces . . . 8
The P^{3}M Algorithm . . . 10
Combining PP for Short-Range and PM for Long-Range . . . 10
The Subtractive Scheme . . . 10
Calculating the Mesh-Force Correction . . . 10
Choosing the Cutoff Radius (r_c) . . . 11
The Switching Function . . . 11
An Expanding Space . . . 12
The Hubble Flow . . . 12
Comoving Coordinates . . . 12
The Equations of Motion . . . 13
An Einstein-de Sitter Universe . . . 13
Initial Conditions . . . 14
The Unperturbed State . . . 14
The Zel'dovich Approximation . . . 15
Validation and Accuracy . . . 17
Conservation of Energy and Momentum . . . 17
The Two-Body Problem and Kepler's Laws . . . 18
Sources of Error . . . 19

Notes on N-Body Simulation

This is a living document—a collection of knowledge that I have gathered while learning about cosmological simulations. It is not a formal text but rather a journal, an attempt to solidify complex concepts by structuring and explaining them in my own way.

This process has been a dual one. Alongside the theoretical exploration, I have been developing a "toy model"—a simple N-body simulation. This hands-on approach allowed me to understand algorithms by implementing them, and to appreciate physical principles by seeing their effects in a virtual universe. The explanations in this document came from the challenges of this practical work.

This is my best effort (and my AI assistant's as well) to present this knowledge in the way that I would have found most helpful at the start of my learning process.

Victor Alamo vialamo@gmail.com

The N-Body Problem

The **N-body problem** is the task of predicting the dynamical evolution of a system composed of N particles that interact through mutual gravitational attraction. Each particle experiences the combined gravitational influence of all others, and because these forces depend on the instantaneous positions of every particle, the motion of any one particle cannot be determined independently.

In Newtonian gravity, the equation of motion for particle i with position vector $\mathbf{x}_i$, velocity $\mathbf{v}_i$, and mass m_i is:

$$m_i \frac{d^2\mathbf{x}_i}{dt^2} = -Gm_i \sum_{\substack{j=1 \\ j\neq i}}^{N} m_j \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|^3}$$

This coupled system of $3N$ second-order differential equations has no general analytic solution for $N > 2$, making it one of the foundational challenges of computational astrophysics.

Direct Summation

The most straightforward numerical method for solving the N-body problem is the **direct-summation algorithm**, which explicitly computes the gravitational force on each particle from every other particle. The procedure for a single time step can be described as follows:

1. **Select a particle**, say particle A.
2. **Loop over all other particles** $(B, C, D, \ldots)$.
3. **Compute the pairwise force** on A from each other particle using Newton's law of gravitation:

   $$\mathbf{F}_{AB} = -G\frac{m_A m_B}{r_{AB}^2}\hat{\mathbf{r}}_{AB}$$

   where $\mathbf{r}_{AB} = \mathbf{x}_A - \mathbf{x}_B$ and $\hat{\mathbf{r}}_{AB} = \mathbf{r}_{AB}/|\mathbf{r}_{AB}|$.
4. **Sum all pairwise forces** to obtain the total force on particle A:

   $$\mathbf{F}_A = \sum_{\substack{B=1 \\ B\neq A}}^{N} \mathbf{F}_{AB}$$

5. **Update** particle A's position and velocity using this total force (via an integration method such as Velocity Verlet).

6. **Repeat** the process for every particle in the system.

Although conceptually simple and physically exact, the direct-summation method is computationally prohibitive for large N. To compute the total force on one particle, we must evaluate $N-1$ pairwise interactions. Doing this for all N particles requires approximately $N(N-1) \approx N^2$ force evaluations per time step.

In computational complexity terms, this corresponds to $O(N^2)$ scaling — meaning that doubling the number of particles multiplies the total computational cost by roughly four. This quadratic growth rapidly becomes intractable: while $N \sim 10^3$ is easily manageable, $N \sim 10^6$ would require on the order of 10^{12} pairwise force evaluations per step.

Because of this steep scaling, the direct method is impractical for cosmological simulations, which often involve millions or billions of particles. To overcome this, we rely on **approximation schemes** — such as the **Particle-Mesh (PM)** and **Particle-Particle Particle-Mesh (P³M)** — that preserve physical accuracy while reducing computational cost from $O(N^2)$ to nearly $O(N \log N)$ or better.

Boundaries and Singularities

Two fundamental problems arise when trying to model gravity in a computer. The first is how to simulate an infinite universe in a finite box, and the second is how to handle the infinite force that occurs when two particles get too close.

Periodic Boundary Conditions

To simulate a small, representative patch of an infinite, uniform universe without having particles react to artificial "walls", simulations employ **periodic boundary conditions**. This method treats the simulation space as a seamless, repeating tile.

A particle exiting one face immediately re-enters from the opposite face. This means that when calculating the force between two particles, the "wrap-around" distance must be considered. We always use the shortest path between the two particles. This is known as the **Minimum Image Convention**, and it ensures that no particle ever feels an artificial "edge of the universe."

Gravitational Softening

Newton's law of gravity, $F \propto 1/r^2$, has a mathematical singularity: as the distance r between two particles approaches zero, the force between them approaches infinity. In a simulation that moves in discrete time steps, these immense forces can cause particles to be catapulted away at unrealistic speeds, completely wrecking the simulation's stability and energy conservation.

To prevent this, we introduce **gravitational softening**. This technique modifies Newton's law of gravity by adding a parameter known as the **softening length**, ϵ (epsilon), to the denominator:

$$F = \frac{Gm_1m_2}{r^2 + \epsilon^2}$$

This simple addition ensures the denominator can never be zero. When particles are far apart, they feel the normal $1/r^2$ force. However, when their separation becomes comparable to or smaller than ϵ, the force is "softened" and stops growing, leveling off at a large but finite value.

A common and effective rule of thumb is to base the softening length on the **mean inter-particle spacing**, d. For a box with side L and N particles, it's calculated as:

$$d = \frac{L}{N^{1/3}}$$

A typical choice for the softening length is then a small fraction of this, such as $\epsilon = d/30$. This ensures that the force is physically accurate for the vast majority of interactions, while the softening only activates during rare, close encounters to prevent numerical catastrophe.

Reference Springel, V. (2005). *The cosmological simulation code GADGET-2. Monthly Notices of the Royal Astronomical Society*, 364(4), 1105-1134. arXiv:astro-ph/0505010. Available at https://arxiv.org/abs/astro-ph/0505010

The Integrator

The Euler Method

To move the particles through time, we need an "integrator"—an algorithm that takes the current state of a particle (its position and velocity) and predicts its state a small moment later. The most intuitive and straightforward approach is the **Euler method**.

The Euler method assumes that the velocity and acceleration are constant over one small time step, Δt. It calculates the force on the particle at its current position to find its acceleration, and then takes a linear step forward.

The update equations are: 1. **Update Position:** $\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_n \Delta t$ 2. **Update Velocity:** $\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \Delta t$

While simple, the Euler method's core assumption is almost always wrong. In a gravitational system, the force is constantly changing as a particle moves, but the Euler method is blind to any changes that occur during the step.

This error, while tiny on each step, is **systematic**. It always pushes the energy in the same direction. Over thousands of steps, this causes a simulated planet to slowly spiral outwards, gaining energy with every orbit until it eventually flies away. This failure to conserve energy makes the Euler method unsuitable for any simulation where long-term stability is important.

Velocity Verlet

The failure of the Euler method shows that a more robust integrator is needed—one that accounts for the fact that forces change *during* a time step. An effective solution is an algorithm called **Velocity Verlet**.

The core idea is to use a more accurate, averaged acceleration to update the velocity. Instead of just using the acceleration from the beginning of the step, it uses the average of the accelerations from the beginning and the end of the step.

The algorithm proceeds in three steps:

1. **Calculate the New Position:** First, advance the position using the current velocity and acceleration.
$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2$$
2. **Calculate the New Acceleration:** With the new position, calculate the new force vector $\mathbf{F}(\mathbf{x}(t + \Delta t))$ and from it, the new acceleration.
$$\mathbf{a}(t + \Delta t) = \frac{\mathbf{F}(\mathbf{x}(t + \Delta t))}{m}$$
3. **Calculate the New Velocity:** Finally, update the velocity using the **average** of the old acceleration $\mathbf{a}(t)$ and the new acceleration $\mathbf{a}(t + \Delta t)$.
$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{\mathbf{a}(t) + \mathbf{a}(t + \Delta t)}{2}\Delta t$$

This final step of averaging the accelerations is the key. It corrects for the systematic drift of the Euler method, and it is what makes Velocity Verlet a **symplectic integrator**. This crucial property is what enables the algorithm to produce a stable and accurate trajectory, conserving energy remarkably well over long periods.

Symplectic Integration

A **symplectic integrator** is an algorithm specifically designed to respect the underlying geometry of physics, a property that allows it to conserve a system's total energy over very long periods. The practical importance of this is best understood by comparing how different integrators handle a simple gravitational problem, like a planet orbiting a star.

- A **non-symplectic** integrator, like the Euler method, consistently makes an error in the same direction. It always "cuts the corner" of the orbit, pushing the planet slightly outwards. These errors add up, causing the planet's energy to systematically increase and its orbit to spiral away.
- A **symplectic** integrator, like Verlet, makes errors that are correlated. On one step, it might slightly overshoot the true orbit, but on a later step, it will undershoot it to compensate. The errors effectively cancel each other out over time.

Instead of a catastrophic spiral, the simulated planet executes a stable "wobble" along the correct orbital path. The shape of the orbit might oscillate slightly, but its average size and, crucially, its average energy, remain correct for millions of steps.

The deeper reason for this remarkable stability lies in a concept from classical mechanics called **phase space**. Phase space is an abstract map where every point represents the complete state of a particle—both its **position** and its **momentum**. For a system where energy is conserved, a fundamental rule known as **Liouville's Theorem** states that the "area" (or volume) of any group of states in phase space must stay constant as the system evolves.

Symplectic integrators are mathematically constructed to **perfectly preserve this phase space volume**. Because they respect this fundamental geometric rule, they are forbidden from having the systematic energy drift that plagues simpler methods. The bounded energy error (the "wobble") is a direct consequence of this property.

This is why symplectic integrators are chosen over the Euler method for any long-term simulation of a conservative system.

The Kick-Drift-Kick Integrator

The standard Velocity Verlet integrator is a powerful tool, but it was designed for a universe with static rules. The introduction of cosmic expansion adds a velocity-dependent term to the equations of motion (the "Hubble drag", explored in a later section). This new term creates a challenge for the standard Verlet algorithm because its symplectic nature is strictly defined for forces that depend only on position, not velocity. To handle this new term gracefully, we adopt a different formulation known as a **Leapfrog** scheme. The most common implementation, the **Kick-Drift-Kick (KDK)** integrator, is the standard choice in cosmological simulations.

The KDK scheme advances the system from a time t to $t + \Delta t$ in three stages:

1. First "Kick" (Velocity Half-Step)

The velocities are "kicked" forward by half a time step using the acceleration from the beginning of the step.

$$\mathbf{v}(t + \tfrac{1}{2}\Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\frac{\Delta t}{2}$$

2. "Drift" (Position Full-Step)

The positions then "drift" for a full time step using the more accurate **mid-step velocity**.

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t + \tfrac{1}{2}\Delta t)\Delta t$$

3. Second "Kick" (Velocity Second Half-Step)

Finally, the new acceleration, $\mathbf{a}(t + \Delta t)$, is computed from the forces at the new positions, $\mathbf{x}(t + \Delta t)$, and the **mid-step velocity**, $\mathbf{v}(t + \frac{1}{2}\Delta t)$. The acceleration is then used to complete the velocity update for the second half of the time step.

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t + \tfrac{1}{2}\Delta t) + \mathbf{a}(t + \Delta t)\frac{\Delta t}{2}$$

While mathematically equivalent to Verlet in simpler cases, this staggered formulation is particularly robust for handling the time-varying and velocity-dependent forces present in a cosmological simulation. The symmetric "kick-force-kick" structure gracefully incorporates these complexities, which is why the KDK leapfrog is the workhorse integrator for nearly all modern cosmological N-body codes.

Reference Tsang, D., Galley, C. R., Stein, L. C., & Turner, A. (2015). *Slimplectic Integrators: Variational Integrators for General Nonconservative Systems.* arXiv:1506.08443. Available at https://arxiv.org/pdf/1506.08443.pdf.

The Particle-Mesh Method

The Particle-Mesh (PM) method is built on a different perspective. Instead of calculating the gravitational pull between every pair of particles, it simplifies the problem by describing the mass distribution on a regular grid. From this "mass map", the gravitational potential and forces can be solved on the grid itself. These are the steps:

1. **Potential calculation:** First, the gravitational potential (Φ) is calculated for the entire grid. The potential is a scalar "landscape" that describes the depth of the gravitational well at every point.
2. **Force calculation:** Second, the force ($\mathbf{F}$) is determined by finding the steepest downhill slope (the gradient) of that potential landscape.

This `Mass -> Potential -> Force` pipeline is the foundation of the PM method. The following sections break down how each part of this process is achieved.

Step 1. Finding the Potential

The process of finding the potential begins by describing the mass distribution on the grid, which then serves as the input for the physical law that governs how that mass creates the potential.

Mass Assignment (NGP)

The first step in this process is **mass assignment**: the procedure for transferring the mass of our continuously positioned particles onto the discrete nodes of the grid.

The simplest and most intuitive way to do this is the **Nearest Grid Point (NGP)** scheme: for each particle, we find the single grid point (or cell center) that it is closest to, and assign the particle's *entire mass* to that one point.

The result is an array representing the mass density field, $\rho_{i,j,k}$. Mathematically, the density in a given cell (i, j, k) is the sum of the masses of all particles within that cell, divided by the cell's volume:

$$\rho_{i,j,k} = \frac{1}{L^3} \sum_{p \in \text{cell}(i,j,k)} m_p$$

Where m_p is the mass of a particle p, and L is the side length of a grid cell.

While NGP is very simple, it can introduce inaccuracies. As we will explore in a later section, more sophisticated schemes like Cloud-in-Cell (CIC) can be used to create a smoother and more accurate density field.

Poisson's Equation

The potential field Φ can be determined from the mass density field, $\rho_{i,j,k}$. The fundamental law linking mass to gravitational potential is **Poisson's Equation**.

$$\nabla^2 \Phi = 4\pi G\rho$$

Where:

- ρ is the mass density grid — the **input**.
- Φ is the gravitational potential field — the **output**.
- G is the gravitational constant.
- ∇^2 (the **Laplacian**) is a mathematical operator that measures how much a function curves around a point—the **net curvature**.

In this equation, mass acts as the source of curvature: where there is mass, the potential bends inward, forming gravitational wells that attract other masses. Where $\rho = 0$, there's no net curvature. This may seem counterintuitive, as the potential field clearly forms a curved, gravitational well even in the empty space around a mass. The key is that the Laplacian, $\nabla^2\Phi$, measures the **net curvature**. In the smooth $1/r$ shape of a potential in empty space, the radial inward bending of the field is perfectly balanced by a natural geometric spreading effect in three dimensions. These two effects cancel each other out, resulting in zero net curvature.

Solving Poisson's equation means finding the global shape of Φ given all the local sources ρ. Doing this directly is computationally expensive, but as we'll see next, the Fast Fourier Transform (FFT) offers an efficient way to compute it by turning this differential problem into simple multiplications in frequency space.

The FFT and the Convolution Theorem

Given the mass density grid, ρ, and the rule connecting it to the potential, Poisson's Equation, the challenge now is to solve it. Calculating the potential at every grid point by summing the influence from all other grid points is a "brute-force" operation known as a **convolution**. It's a slow, computationally expensive task.

Fortunately, there is a more efficient mathematical tool that can solve this problem: the **Fast Fourier Transform (FFT)**. This algorithm is used to take advantage of the **Convolution Theorem**.

The Fourier Transform The **Fourier Transform** is a mathematical operation that rewrites any spatial field in terms of its **spatial frequencies** — the underlying wave patterns that make it up.

Suppose we have a density field $\rho(\mathbf{x})$ defined across space. Instead of describing it point by point, we can express it as a sum of smooth oscillating patterns — *plane waves* — each with its own wavelength, direction, and amplitude. The Fourier Transform tells us **how each wave** contributes to the total field.

Formally, it is written as:

$$\hat{\rho}(\mathbf{k}) = \int \rho(\mathbf{x}) e^{-i\mathbf{k}\cdot\mathbf{x}} d^3x$$

Here, $\mathbf{x}$ represents position in real space, and $\mathbf{k}$ is the **wavevector**, describing one of those plane waves. Each component (k_x, k_y, k_z) measures how rapidly the field oscillates along that direction, while its magnitude

$$k = |\mathbf{k}| = \frac{2\pi}{\lambda}$$

tells us the *spatial frequency*.

The result of the transform, $\hat{\rho}(\mathbf{k})$, is a **complex number**. Its magnitude $|\hat{\rho}(\mathbf{k})|$ gives the *amplitude*, and its phase $\arg(\hat{\rho}(\mathbf{k}))$ specifies the *offset* — where the wave's peaks and troughs occur in space.

Thus, while $\rho(\mathbf{x})$ is a **real-valued function of position**, $\hat{\rho}(\mathbf{k})$ is a **complex-valued function of wavevector**. They describe the same field, but from two complementary perspectives: one in **Real space**, one in **Frequency (or k-) space**. This dual representation is very useful in simulations.

> In what follows, we'll use the shorthand notation ρ_k and Φ_k to denote the discrete Fourier-transformed fields, corresponding to $\hat{\rho}(\mathbf{k})$ and $\hat{\Phi}(\mathbf{k})$ in the continuous case, but defined only at the discrete $\mathbf{k}$ values of our simulation grid.

The Convolution Theorem This theorem is the heart of the entire PM method. It states:

> A slow and complicated **convolution** in real space becomes a fast and simple element-by-element **multiplication** in frequency space.

This allows us to replace the slow, brute-force calculation with a faster three-step process:

1. **Transform to Frequency Space:** We use the **FFT** to transform our mass grid, ρ, into its frequency representation, ρ_k. A convenient property of the FFT is that it automatically treats the finite grid as if it were periodic. It represents the data as a sum of simple waves that fit perfectly end-to-end within the box, which is mathematically equivalent to assuming the grid repeats infinitely like a tiled pattern.

   To compute the gravitational potential, we also need the transform of the function that describes how a unit mass influences space—the **Green's function**. Formally, the Green's function, $G(\mathbf{r})$, is defined as the solution to Poisson's equation for a unit point source:

   $$\nabla^2 G(\mathbf{r}) = 4\pi G\delta(\mathbf{r}).$$

   Where $\delta(\mathbf{r})$ is the Dirac delta function, which represents an idealized point source. In three dimensions, this gives $G(\mathbf{r}) = -G/|\mathbf{r}|$. This mathematical kernel acts as the system's response to a point mass, linking the density field to the potential through Poisson's equation.

   When we move to frequency space, derivatives become multiplications by $-k^2$, so the corresponding frequency-space form of the Green's function is

   $$\mathcal{F}\{\nabla^2 G(\mathbf{r})\} = -k^2 G_k$$

   $$\mathcal{F}\{4\pi G\delta(\mathbf{r})\} = 4\pi G$$

   $$G_k = -\frac{4\pi G}{k^2}.$$

   This is the function we use to compute the potential in the Fourier domain. This function has a mathematical singularity at the $k = 0$ **mode**. This mode (also known as the DC component) represents the **average potential** of the entire simulation box.

However, since physical forces depend only on the **gradient** of the potential ($\mathbf{F} = -\nabla\Phi$), not its absolute value, this average potential is physically arbitrary. To avoid the division-by-zero, we can redefine the $k = 0$ component of the potential to an arbitrary value (usually zero). This is a standard procedure that makes the calculation well-defined without affecting the final forces.

2. **Multiply:** In frequency space, the Poisson equation becomes an element-wise multiplication:

$$\Phi_k = G_k \cdot \rho_k$$

3. **Transform Back:** We take the resulting potential in frequency space, Φ_k, and use the **Inverse Fast Fourier Transform (IFFT)** to convert it back into the real-space potential grid, Φ, that we wanted.

By taking this detour through frequency space, we replace a slow algorithm that scales as $O(M^6)$ (for M grid cells) with one that scales as $O(M^3 \log M)$. This is what makes the Particle-Mesh method convenient, enabling simulations with millions or billions of particles.

Step 2. From Potential to Force

Now that we have the potential grid, Φ, we can calculate the force grid. The physical relationship is universal: force is the negative gradient of the potential.

$$\mathbf{F} = -\nabla\Phi$$

On a discrete grid, we can't take a true derivative. We approximate it using a **finite difference**. A common and accurate method is the **central difference**, which calculates the slope at a point by looking at the values of its neighbors on either side. For the x-component of the force at grid cell (i, j, k), the formula is:

$$F_{x,i,j,k} \approx -\frac{\Phi_{i+1,j,k} - \Phi_{i-1,j,k}}{2L}$$

With the force calculated at every point on the grid, the final step is to **interpolate** this force back to each particle's continuous position. This is done using the same scheme we used for mass assignment (e.g., NGP or CIC), completing the Particle-Mesh calculation.

Reference Breton, Michel-Andrès. (2024). *PySCo: A fast Particle-Mesh N-body code for modified gravity simulations in Python.* arXiv:2410.20501. Available at https://export.arxiv.org/abs/2410.20501

Advanced Interpolation

The Flaws of Nearest Grid Point (NGP)

In a previous section, we introduced the Nearest Grid Point (NGP) scheme as the simplest way to assign mass to the grid. While its simplicity is appealing, it comes at a significant cost to the simulation's accuracy. The "blocky," pixelated density field it creates leads to an equally blocky and unphysical force field.

The primary flaw of NGP is that the force a particle feels is **discontinuous**. A real gravitational field is smooth and changes continuously with position. The jerky, stepwise force from an NGP grid is a poor and unphysical approximation. This leads to several significant problems:

1. **Poor Energy Conservation:** This is the most damaging consequence. Symplectic integrators like Velocity Verlet can only conserve energy if the force is the smooth gradient of a potential. The sudden "jumps" in force at the cell boundaries introduce small, systematic errors into the integration. These errors accumulate over time, causing the total energy of the simulation to **drift** upwards or downwards, rather than just oscillating around the true value.

2. **Violation of Newton's Third Law:** While the total momentum of the system may be globally conserved, the force between any specific pair of particles is not guaranteed to be equal and opposite. The force on particle A depends only on which cell it's in, and the force on particle B depends only on which cell *it's* in. This crude mediation by the grid breaks the pairwise symmetry required for good energy conservation.

3. **Grid-Imposed Artifacts:** The force field has an artificial, grid-like pattern. Particles can feel an unphysical pull along the grid axes (x and y) that is stronger than the pull along the diagonals. This can cause particles to artificially cluster along grid lines, a distracting and inaccurate artifact of the method.

Because of these flaws, NGP is rarely used in simulations where accuracy is a priority. To achieve the stable, energy-conserving behavior we need, we must adopt a smoother method for connecting the particles to the grid, which leads us to the Cloud-in-Cell scheme.

Cloud-in-Cell (CIC)

To achieve a stable simulation that conserves energy, we need a smooth way to connect the particles to the grid. The standard method for this is the **Cloud-in-Cell (CIC)** interpolation scheme.

Particles as Clouds

Instead of treating each particle as an infinitesimal point, the CIC method imagines each particle as a small, **cubic "cloud"** of mass, the same size as a grid cell. As this particle-cloud moves through the simulation space, it naturally overlaps with the **eight** nearest grid points that form the corners of the cell it currently occupies.

The mass of the particle is then distributed, or "splatted," onto these eight grid points. The amount of mass assigned to each point is simply proportional to the **volume of overlap** between the particle's cloud and the region surrounding each grid point. This is a form of **trilinear interpolation**. A particle in the exact center of a cubic cell would distribute 12.5% of its mass to each of the eight corners. A particle mostly in one corner of a cell would give most of its mass to that corner's node.

This process results in a much smoother and more physically realistic mass density grid. A small movement by a particle leads to a small, continuous change in the mass distribution on the grid, completely eliminating the sudden "jumps" of the NGP method.

At its core, CIC is a linear interpolation scheme. Higher-order schemes (like TSC or PCS) exist and provide smoother forces, but are not in the scope of this text.

Symmetric Interpolation

A smooth mass distribution is only half the story. The true magic of CIC, and the reason it conserves energy, lies in its perfect symmetry.

After the forces have been calculated on the grid, we must interpolate them back to the particle's continuous position. The rule for energy conservation is that the **force interpolation scheme must be consistent with the mass assignment scheme**.

CIC follows this rule perfectly. The force on the particle is calculated by taking a weighted average of the forces from the **same eight grid points**, using the **exact same volume-based weights** that were used to distribute the mass.

This symmetry between "splatting" the mass and "gathering" the force ensures that Newton's third law ($\mathbf{F}_{ij} = -\mathbf{F}_{ji}$) is precisely obeyed for any pair of particles, even though their interaction is being mediated by the grid. Because the forces are perfectly reciprocal, the force field is numerically conservative.

When this conservative force is fed into a symplectic integrator, the system's total energy is conserved remarkably well. The systematic energy *drift* seen with NGP is transformed into the small, bounded *oscillation* expected from a high-quality simulation. While slightly more complex to implement, the improvement in accuracy and stability makes CIC the standard choice for most modern Particle-Mesh codes.

Implementation: "Splatting" Mass and "Gathering" Forces

The conceptual idea of treating particles as "clouds" translates into a clean, two-part algorithm. In simulation jargon, these two parts are often called **"splatting"** (distributing the particle mass onto the grid) and **"gathering"** (interpolating the force from the grid back to the particle).

The key to energy conservation is that these two operations must be perfectly symmetric, using the exact same weights for both processes.

For simplicity, the following explanation is for a 2D case.

The "Splat" Step: Mass Assignment

This process is performed for every particle to create the final mass density grid, ρ.

1. Find the Reference Grid Point and Fractional Position

First, for a given particle p with position $\mathbf{x}_p = (x_p, y_p)$, we find the integer index of the grid point at its "bottom-left," denoted (i, j). We then find the particle's fractional position within that cell, (δ_x, δ_y), where both values range from 0 to 1. Let L be the side length of a grid cell.

The reference grid index is found using the floor function, $\lfloor \cdot \rfloor$:

$$i = \lfloor x_p/L \rfloor$$

$$j = \lfloor y_p/L \rfloor$$

The fractional position within the cell is then:

$$\delta_x = (x_p/L) - i$$

$$\delta_y = (y_p/L) - j$$

2. Calculate the Area Weights

Next, we calculate four weights based on these fractional positions. Each weight corresponds to the fractional area of the particle's "cloud" that overlaps with the four surrounding grid cells located at (i, j), $(i + 1, j)$, $(i, j + 1)$, and $(i + 1, j + 1)$.

The weights for the corners are:

$$w_{i,j} = (1 - \delta_x)(1 - \delta_y)$$

$$w_{i+1,j} = \delta_x(1 - \delta_y)$$

$$w_{i,j+1} = (1 - \delta_x)\delta_y$$

$$w_{i+1,j+1} = \delta_x\delta_y$$

Notice that these four weights always sum to 1.

3. Distribute the Mass

Finally, we add the mass of the particle, m_p, scaled by the appropriate weight, to each of the four corresponding grid points. This process is repeated for all particles in the simulation. The total mass on a given grid point is the sum of the contributions from all particles whose "clouds" overlap it.

The contribution from a single particle p to the mass at each of the four nodes is:

$$\Delta M_{i,j} = m_p \cdot w_{i,j}$$

$$\Delta M_{i+1,j} = m_p \cdot w_{i+1,j}$$

$$\Delta M_{i,j+1} = m_p \cdot w_{i,j+1}$$

$$\Delta M_{i+1,j+1} = m_p \cdot w_{i+1,j+1}$$

To get the final mass density grid, ρ, the total mass accumulated at each node is divided by the cell area, L^2. All grid indices are taken modulo the grid size to correctly handle the periodic boundaries.

The "Gather" Step: Force Interpolation

This step occurs after the forces have been calculated on the grid (creating an acceleration field, $\mathbf{a}_{\text{grid}}$) and is the mirror image of the splatting process.

To find the force on a particle, we use the **exact same** indices and weights we would calculate for it in the splatting step. We then perform a weighted average of the acceleration values from the four surrounding grid points to find the acceleration at the particle's precise location, $\mathbf{a}_p$.

Let the acceleration field on the grid be $\mathbf{a}_{i,j} = (a_{x,i,j}, a_{y,i,j})$ and the four CIC weights for a given particle be $w_{i,j}$, $w_{i+1,j}$, $w_{i,j+1}$, and $w_{i+1,j+1}$.

The x-component of the interpolated acceleration for the particle, $a_{x,p}$, is calculated as:

$$a_{x,p} = a_{x,i,j} \cdot w_{i,j} + a_{x,i+1,j} \cdot w_{i+1,j} + a_{x,i,j+1} \cdot w_{i,j+1} + a_{x,i+1,j+1} \cdot w_{i+1,j+1}$$

The y-component, $a_{y,p}$, is calculated in the exact same way using the y-components of the grid acceleration field.

The final force on the particle, $\mathbf{F}_p$, is its mass, m_p, times this interpolated acceleration vector:

$$\mathbf{F}_p = m_p \mathbf{a}_p$$

This symmetric "Splat-Gather" procedure ensures that the forces are numerically conservative, which is the fundamental reason why CIC allows a symplectic integrator to conserve energy over long periods.

Reference Bagla, J. S., & Padmanabhan, T. (2004). *Cosmological N-Body Simulations.* arXiv:astro-ph/0411730. Available at https://arxiv.org/pdf/astro-ph/0411730.pdf

The P³M Algorithm

Combining PP for Short-Range and PM for Long-Range

We have seen that the Particle-Mesh (PM) method is efficient for calculating the gravitational field of a large number of particles. However, its speed comes at the cost of accuracy at small scales. The grid is good at capturing the overall "blurry" shape of the gravitational field, but it's innacurate at resolving the sharp, fine details of the force between two particles that are very close to each other. This inaccuracy at short ranges is the primary weakness of the pure PM method.

On the other hand, the direct Particle-Particle (PP) calculation is the exact opposite. While it is perfectly accurate at all scales, its weakness, is that its $O(N^2)$ complexity makes it too slow for a large number of particles.

This presents a classic trade-off: speed or accuracy. The **Particle-Particle Particle-Mesh (P³M)** algorithm provides an elegant solution by combining both methods, using each one only where it excels.

The P³M method splits the force calculation into two parts based on a **cutoff radius**, r_c:

1. **Long-Range Force (PM):** The smooth, gentle pull from all the **distant** particles (those farther than r_c) is calculated efficiently using the Particle-Mesh method.
2. **Short-Range Force (PP):** The sharp, strong force from the few **nearby** particles (those closer than r_c) is calculated precisely using the direct Particle-Particle method.

The Subtractive Scheme

Simply adding these two forces together would be incorrect, as the PM method already includes an inaccurate estimate of the short-range forces. Instead, we use the PP calculation to *correct* the PM force at short distances. This is often done with a **subtractive scheme**:

$$\mathbf{F}_{\text{total}} = \mathbf{F}_{\text{PM}} + \left(\mathbf{F}_{\text{PP}}^{\text{short}} - \mathbf{F}_{\text{PM}}^{\text{short}}\right)$$

The process is straightforward:

1. First, we calculate the baseline $\mathbf{F}_{\mathbf{PM}}$ for all particles. This gives us the correct long-range force everywhere but an incorrect "blurry" force for nearby pairs.
2. Then, for any pair of particles closer than the cutoff radius, we calculate the **true, sharp force** between them, $\mathbf{F}_{\mathbf{PP}}^{\mathbf{short}}$.
3. We also calculate an approximation of the **blurry, inaccurate force** that the PM method produced for that same pair, $\mathbf{F}_{\mathbf{PM}}^{\mathbf{short}}$.
4. Finally, we subtract the inaccurate mesh force and add the correct direct force. This effectively replaces the blurry grid force with the sharp, accurate PP force, but only where it matters—at short distances.

By dividing the problem this way, P³M leverages the strengths of both methods. It uses the fast PM algorithm for the vast majority of interactions (the thousands of weak pulls from distant particles) and reserves the slow but accurate PP algorithm only for the few critical interactions between close neighbors. The result is a simulation that is nearly as fast as a pure PM code but nearly as accurate as a pure PP code—the true best of both worlds.

Calculating the Mesh-Force Correction

To implement the subtractive scheme, we need a mathematical function for $\mathbf{F}_{\text{PM}}^{\text{short}}$ that approximates the "blurry" force produced by the grid at short distances. We can't get this from the final grid itself, as it contains the combined influence of all particles.

Instead, we model this effect with a standard gravitational force formula that has been **softened** with a special parameter, ϵ_{PM}, chosen specifically to mimic the resolution of the Particle-Mesh grid.

The vector force that approximates the mesh's influence between two particles with masses m_1 and m_2 is given by:

$$\mathbf{F}_{\text{PM}}^{\text{short}} = \frac{Gm_1m_2\mathbf{r}}{\left(r^2 + \epsilon_{\text{PM}}^2\right)^{3/2}}$$

The terms in this formula are: * $\mathbf{r}$ is the vector separating the two particles. * r is the magnitude of that vector, $r = \|\mathbf{r}\|$. * G, m_1, m_2 are the gravitational constant and the particle masses. * ϵ_{PM} is the crucial term: a **softening length** specifically chosen to match the grid's resolution. A standard and effective choice is to set this value to be proportional to the grid cell length, L. For example:

$$\epsilon_{\text{PM}} \approx 0.5 \cdot L$$

This formula creates a force that is significantly weakened at short distances (when $r \lesssim \epsilon_{\text{PM}}$), which successfully mimics the behavior of the full PM/FFT calculation. By subtracting this specific force in the correction step, we effectively cancel out the grid's primary error at short range.

Choosing the Cutoff Radius (r_c)

The choice of the cutoff radius, r_c, is a crucial tuning parameter in a P³M simulation. It represents a trade-off between accuracy and computational speed.

- A **small** cutoff radius means the fast PM method handles most of the work, but we risk losing accuracy if the cutoff is smaller than the region where the PM force is unreliable.
- A **large** cutoff radius ensures high accuracy at short ranges, but it forces the slow PP calculation to do much more work, which can bog down the entire simulation.

The optimal choice is not arbitrary; it is fundamentally linked to the resolution of the Particle-Mesh grid. The PM method's accuracy degrades significantly at distances smaller than about 2 to 3 grid cell sizes. Therefore, the cutoff radius must be large enough to ensure the accurate PP method is used throughout this entire "inaccurate zone."

A standard and robust rule of thumb is to set the cutoff radius to be a few times the grid cell length, L:

$$r_c \approx 2.5 \cdot L$$

This choice guarantees that the sharp, correct PP force is used wherever the PM force is most likely to fail. The primary parameter that is usually tuned is the grid size itself; once the grid size is chosen, the cutoff radius is set accordingly to maintain this balance.

The Switching Function

The subtractive scheme is a powerful way to correct for the short-range errors of the Particle-Mesh method. However, using a hard cutoff radius—where the correction is fully active if $r < r_c$ and instantly zero if $r \geq r_c$—can create a small, abrupt "jolt" in the force. This discontinuity, however small, can introduce numerical errors and impact the long-term energy conservation of the simulation.

To achieve the highest accuracy, we must ensure the total force is a perfectly smooth function at all distances. This is accomplished by introducing a **switching function**, $S(r)$, that smoothly "fades out" the short-range correction as the particle separation, r, approaches the cutoff radius, r_c.

The total force is then calculated as:

$$\mathbf{F}_{\text{total}} = \mathbf{F}_{\text{PM}} + S(r) \cdot \left(\mathbf{F}_{\text{PP}}^{\text{short}} - \mathbf{F}_{\text{PM}}^{\text{short}}\right)$$

The switching function $S(r)$ operates over a small **transition zone**, typically defined between a starting radius, r_{start}, and the cutoff radius, r_c. It has the following properties:

1. For $r \leq r_{\text{start}}$, the function is $S(r) = 1$. The correction is fully applied.
2. For $r \geq r_c$, the function is $S(r) = 0$. The correction is fully turned off.
3. In the transition zone, $r_{\text{start}} < r < r_c$, the function smoothly decreases from 1 to 0.

To ensure the force changes perfectly smoothly, the *derivative* of the switching function should also be zero at the start and end of the transition. A standard and effective way to achieve this is with a cubic polynomial.

First, we define a normalized distance, x, that goes from 0 to 1 across the transition zone:

$$x = \frac{r - r_{\text{start}}}{r_c - r_{\text{start}}}$$

Then, a polynomial that satisfies all the smoothness conditions is:

$$S(x) = 2x^3 - 3x^2 + 1$$

Using this function to taper the correction term eliminates the unphysical jolt at the cutoff. It creates a continuous and differentiable total force, which allows the symplectic integrator to perform optimally and leads to superior long-term energy conservation.

Reference Shirokov, A., & Bertschinger, E. (2005). *GRACOS: Scalable and Load Balanced P3M Cosmological N-body Code.* arXiv:astro-ph/0505087. Available at https://arxiv.org/abs/astro-ph/0505087

An Expanding Space

Up to this point, our simulation has taken place in a static box. This is a good approximation for a star cluster or a single galaxy, but it is fundamentally wrong for a cosmological simulation. Our universe is not static; it is expanding. To accurately model the formation of structure, we must incorporate this expansion into our simulation.

This is achieved by switching from familiar "proper" coordinates to a more abstract but powerful system called **comoving coordinates**. Instead of tracking particles in a fixed box, we track them on a virtual grid that expands along with the universe itself.

The Hubble Flow

The dominant motion in the universe is the cosmic expansion, a phenomenon described by the **Hubble-Lemaître Law**. This law states that, on average, every galaxy is moving away from every other galaxy. The farther away a galaxy is, the faster it appears to recede. This is not a motion *through* space, but rather the expansion *of* space itself. This uniform expansion is the **Hubble Flow**. The velocity of this recession, $\mathbf{v}_{\text{Hubble}}$, for an object at a proper distance $\mathbf{r}$ is given by:

$$\mathbf{v}_{\text{Hubble}}(t) = H(t)\mathbf{r}(t)$$

Where $H(t)$ is the Hubble parameter at time t. This flow is the background upon which all other motions are superimposed.

Comoving Coordinates

Tracking particles whose primary motion is this rapid expansion is computationally difficult. It's much easier to factor out the expansion. We do this by defining a **scale factor**, $a(t)$, which describes the relative size of the universe at any time t. By convention, $a = 1$ today. In the past, a was smaller.

We can now define two types of coordinates: * **Proper Coordinates (r):** The real, physical distance between two objects that you would measure with a ruler at time t. This distance grows as the universe expands. * **Comoving Coordinates (x):** The coordinates of an object on our virtual, expanding grid. If an object is moved *only* by the Hubble Flow, its comoving coordinates **do not change**.

The relationship between them is simple:

$$\mathbf{r}(t) = a(t)\mathbf{x}$$

A particle's true velocity is a combination of the Hubble Flow and its own, small-scale motion relative to the expanding grid. This local motion, caused by the gravitational pull of nearby structures, is called the **peculiar velocity**, $\mathbf{v}_{\text{pec}}$.

The Equations of Motion

To understand how the equations of motion change, we start with the standard physical law in **proper coordinates**: a particle's physical acceleration is equal to the true gravitational acceleration at its location.

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{g}_{\text{proper}}(\mathbf{r})$$

Here, $\mathbf{g}_{\text{proper}}(\mathbf{r})$ is the "real" gravitational acceleration created by the physical distribution of matter in the expanding universe.

Our goal is to rewrite this equation using **comoving coordinates**, $\mathbf{x}$, which are related by $\mathbf{r}(t) = a(t)\mathbf{x}$. After performing the necessary calculus to account for the fact that the scale factor $a(t)$ is changing over time, we arrive at the new equation of motion for a particle's comoving acceleration, $\frac{d^2\mathbf{x}}{dt^2}$:

$$\frac{d^2\mathbf{x}}{dt^2} = \frac{\mathbf{g}_{\text{comoving}}(\mathbf{x})}{a^3} - 2H(t)\frac{d\mathbf{x}}{dt}$$

Let's break down these two terms, which are the fundamental modifications needed for a cosmological simulation.

- **Modified Gravity:** The first term, $\frac{\mathbf{g}_{\text{comoving}}(\mathbf{x})}{a^3}$, represents the force of gravity. The term $\mathbf{g}_{\text{comoving}}(\mathbf{x})$ is the gravitational acceleration that our force solvers (like PM or PP) calculate—it's the acceleration that would exist in a *static* universe if the particles were at their current comoving positions. The division by the scale factor cubed, a^3, is the crucial cosmological correction. As the universe expands by a factor of a, the volume of any given region increases by a^3. This dilutes the physical density of matter as $\rho \propto 1/a^3$. Since gravity is sourced by density, its strength weakens accordingly, and this term correctly captures that effect.
- **Hubble Drag:** The second term, $-2H(t)\frac{d\mathbf{x}}{dt}$, is a new velocity-dependent "friction" term. The term $H(t)$ is the Hubble parameter ($\frac{1}{a}\frac{da}{dt}$), and $\frac{d\mathbf{x}}{dt}$ is the particle's **peculiar velocity** (its local motion relative to the expanding grid). This "Hubble drag" acts to slow down these peculiar velocities. In an expanding universe, a particle's local motion is constantly being damped by the stretching of space itself.

By solving this new equation of motion, our simulation correctly captures the delicate interplay between the cosmic expansion that tries to pull everything apart and the force of gravity that tries to pull everything together.

An Einstein-de Sitter Universe

For a simulation to be physically meaningful, it must be based on a self-consistent cosmological model. The simplest and most classic model for a matter-dominated universe is the **Einstein-de Sitter (EdS)** model. This is a specific solution to Einstein's Friedmann equations that describes a flat, expanding universe containing only matter and no cosmological constant:

$$H(t)^2 = \frac{8\pi G}{3}\rho(t)$$

In an EdS universe, the expansion of space is constantly being decelerated by gravity. This physical reality is described by a simple power-law relationship between the scale factor, a, and cosmic time, t:

$$a(t) \propto t^{2/3}$$

From this, the Hubble parameter,

$$H(t) = \frac{1}{a(t)}\frac{da(t)}{dt},$$

is also a simple function of time:

$$H(t) = \frac{2}{3t}$$

Critical Density and Model Consistency

The defining feature of a flat universe is that its average density, $\rho(t)$, is equal to a special value known as the **critical density**, $\rho_c(t)$. The critical density is the precise density required to halt the cosmic expansion after an infinite amount of time, and it is defined by the Hubble parameter and the gravitational constant, G:

$$\rho_c(t) = \frac{3H(t)^2}{8\pi G}$$

For our simulation to be a consistent representation of an EdS universe, the mean density of our simulation box must be equal to this critical density at all times. The mean density of the box is the total mass, M_{total}, divided by the proper (physical) volume, $V = (aL)^3$:

$$\rho_{mean}(t) = \frac{M_{total}}{(a(t)L)^3}$$

By equating $\rho_{mean} = \rho_c$ and substituting the EdS relations for $a(t)$ and $H(t)$, we find that the simulation parameters are not independent but must be linked by a **consistency relation**.

Natural Units

To simplify the implementation, it is standard practice to work in a system of **natural units** where key quantities are set to 1. A common choice for N-body simulations is to set: * The total mass of the system: $M_{total} = 1$ * The comoving side length of the box: $L = 1$ * The present-day scale factor: $a(t_{today}) = 1$

With these choices, the consistency relation is no longer a check but is used to *define* the value of the gravitational constant required for the simulation. Equating the mean and critical densities in this unit system fixes the value of G:

$$G = \frac{3H(t)^2(a(t)L)^3}{8\pi M_{total}} = \frac{L^3}{6\pi M_{total}} = \frac{1}{6\pi}$$

By using this specific value for G, we ensure that the strength of gravity in our simulation is perfectly balanced against the expansion rate, allowing for the realistic, hierarchical growth of structure.

Reference Springel, V. (2005). The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4), 1105-1134. Available at: https://arxiv.org/abs/astro-ph/0505010

Initial Conditions

The outcome of a simulation is critically dependent on its starting point. We cannot simply place particles randomly in a box; to model our universe, we must create a snapshot that reflects the state of the cosmos at a very early time. This snapshot needs to account for the primordial density fluctuations that acted as the seeds for all future structure.

The Unperturbed State

To represent the nearly uniform matter distribution of the early universe, we begin by placing particles on a perfect, uniform cubic lattice. For a simulation with N particles in a cubic box of side length L, the initial grid position, $\mathbf{x}_{\text{grid}}$, of a particle is determined by its integer indices (i, j, k).

The position is calculated by determining the number of particles per side, N_s, the spacing between them, d, and then placing each particle at the center of its virtual cubic cell:

The number of particles per side is:

$$N_s = N^{1/3}$$

The spacing between grid points is:

$$d = \frac{L}{N_s}$$

The position vector, $\mathbf{x}_{\text{grid}}$, for the particle at indices (i, j, k) is then given by its components:

$$x = \left(i + \frac{1}{2}\right) d$$

$$y = \left(j + \frac{1}{2}\right) d$$

$$z = \left(k + \frac{1}{2}\right) d$$

Where the indices i, j, and k each run from 0 to $N_s - 1$. The addition of $1/2$ ensures that each particle is placed in the center of its cell, rather than at the corner.

The Zel'dovich Approximation

The real universe was not perfectly uniform. To create the seeds of galaxies and clusters, we must apply small, correlated **perturbations** to our particle lattice. The standard method for this is the **Zel'dovich Approximation**, which generates a smooth displacement field to "nudge" each particle from its perfect grid position.

It is crucial to understand the role of the Zel'dovich Approximation in this context. Although it is an analytical theory that describes the linear growth of structure in the early universe, we do not use it to evolve the particles during our simulation. Instead, we use its time-dependent nature *only once* to generate a single, self-consistent snapshot of the universe at our chosen start time, $t_{initial}$. This snapshot provides both the initial particle positions and their corresponding initial peculiar velocities. From that moment forward, the N-body simulation takes over, calculating the full, non-linear evolution of these particles on its own.

Mathematically, the Zel'dovich Approximation is an application of **first-order Lagrangian perturbation theory (1LPT)**. For higher accuracy more advanced schemes such as **second-order Lagrangian perturbation theory (2LPT)** are often employed, but not covered in this text.

The Growth Factor

Although we only use it to generate a single initial snapshot, the Zel'dovich Approximation is fundamentally a dynamic theory in which the displacement field, $\mathbf{\Psi}$, is not constant. As the universe evolves, the tiny initial overdensities attract more matter, causing the perturbations to grow stronger. In the linear regime, the spatial pattern of the displacement field remains fixed, while its amplitude grows over time. This growth is described by a single function of time, the **linear growth factor**, $D(t)$.

The full, time-dependent displacement field can therefore be written as:

$$\mathbf{\Psi}(\mathbf{x}, t) = D(t)\mathbf{\Psi}_0(\mathbf{x})$$

Here, $\mathbf{\Psi}_0(\mathbf{x})$ is the primordial displacement pattern at some reference time (conventionally, today, where $D = 1$), and $D(t)$ scales this entire pattern up or down depending on the cosmic epoch. In the simple Einstein-de Sitter universe we are modeling, the growth factor is conveniently proportional to the scale factor, $D(t) \propto a(t)$.

This separability is incredibly powerful. It means we only need to compute the complex spatial pattern, $\mathbf{\Psi}_0(\mathbf{x})$, once. The state of the universe at any early time is then known simply by scaling this pattern by the appropriate value of $D(t)$.

Generating the Displacement Pattern

The process of generating the spatial pattern, $\mathbf{\Psi}_0(\mathbf{x})$, begins in Fourier space with the **power spectrum**, $P(k)$. The power spectrum is the statistical recipe for our universe's initial conditions, specifying the amplitude of density fluctuations at different spatial scales, k. While more complex options exist, for simplicity, we can use a power-law form:

$$P(k) = Ak^n$$

Where A is a normalization constant and n is the spectral index, which determines the character of the fluctuations. In practice, however, A and n are chosen so that $P(k)$ matches the **ΛCDM (Lambda-Cold Dark Matter)** cosmological power spectrum, which is derived from precise observations of the Cosmic Microwave Background and large-scale structure surveys.

The spectral index, n, is the most important term for defining the *character* of the initial cosmic structure, controlling the balance of power between large-scale (low frequency, k) and small-scale (high frequency, k) fluctuations.

In cosmology, the special "flat" or **scale-invariant** spectrum is defined by a spectral index of $n = 1$. This case, known as the Harrison-Zel'dovich spectrum, represents a universe where the initial fluctuations have the same strength on all physical scales. All real-world spectra are described by how they "tilt" away from this baseline.

- If $n = 1$, we have a **scale-invariant** spectrum. This is the theoretical "white noise" or baseline for cosmology.
- If $n < 1$, we have a **"red-tilted"** spectrum. There is more power in large-scale (low k) fluctuations and less power in small-scale (high k) ones, compared to the scale-invariant case. This results in a universe with large, gentle, rolling waves of density.
- If $n > 1$, we have a **"blue-tilted"** spectrum. There is less power on large scales and more power on small scales, compared to the scale-invariant case.

Observations of the early universe show that our cosmos has a **"red-tilted"** spectrum, with a spectral index n very close to 1 (specifically, $n \approx 0.96$). This means the initial density ripples were slightly stronger on larger scales than on small ones, providing the seeds for the vast cosmic web we see today.

The normalization constant, A, sets the overall **strength** or **amplitude** of the density fluctuations across all scales. A larger value of A would correspond to a lumpier, more violently-collapsing early universe, while a smaller value would lead to a smoother universe where structures form more slowly.

To generate the displacement pattern, $\mathbf{\Psi}_0(\mathbf{x})$, we use the following steps:

1. **Generate a random field in Fourier space.** We start by creating a grid of random complex numbers, $\delta(\mathbf{k})$, that satisfies **conjugate symmetry**, $\delta(\mathbf{k}) = \delta^*(-\mathbf{k})$, to ensure the final field in real space is real-valued.

2. **Shape the random field according to the power spectrum.** Each Fourier mode is then scaled so that its amplitude follows the desired **power spectrum** $P(k)$:

$$\delta_\rho(\mathbf{k}) = \delta(\mathbf{k})\sqrt{P(k)}$$

3. **Compute the gravitational potential and displacement field.** From Poisson's equation, the gravitational potential $\hat{\Phi}(\mathbf{k})$ is related to the density field by $\hat{\Phi}(\mathbf{k}) \propto -\delta_\rho(\mathbf{k})/k^2$. The displacement field is proportional to the gradient of this potential. In Fourier space, taking a gradient corresponds to multiplying by $i\mathbf{k}$:

$$\hat{\mathbf{\Psi}}_0(\mathbf{k}) \propto i\mathbf{k}\frac{\delta_\rho(\mathbf{k})}{k^2}$$

4. **Transform back to real space.** Finally, we apply the inverse Fourier transform to recover the displacement pattern in real space:

$$\mathbf{\Psi}_0(\mathbf{x}) = \mathcal{F}^{-1}\{\hat{\mathbf{\Psi}}_0(\mathbf{k})\}$$

Applying the Displacements and Velocities

With the spatial pattern $\mathbf{\Psi}_0(\mathbf{x})$ calculated, we can now set the initial state of our simulation at its starting time, $t_{initial}$.

The final initial position of each particle is its grid position plus the displacement field, scaled by the growth factor at the start time:

$$\mathbf{x}_{\text{final}}(t_{initial}) = \mathbf{x}_{\text{grid}} + D(t_{initial})\mathbf{\Psi}_0(\mathbf{x}_{\text{grid}})$$

This initial value of the growth factor is determined by the overall amplitude of the power spectrum. This amplitude is typically normalized by observations of large-scale structure today, which by convention sets the growth factor at the present day to one, $D(\text{today}) = 1$. Accordingly, the initial growth factor for the Einstein-de Sitter model, where $D(t) \propto a(t)$, is simply equal to the initial scale factor, $D(t_{initial}) = a_{initial}$.

The initial "peculiar" velocity of a particle (its motion on top of the Hubble flow) is the time derivative of its comoving displacement. It is therefore proportional to the displacement field itself, but scaled by the *rate of change* of the growth factor:

$$\mathbf{v}_{\text{pec}}(t_{initial}) = \left.\frac{dD(t)}{dt}\right|_{t_{initial}} \mathbf{\Psi}_0(\mathbf{x}_{\text{grid}})$$

In the linear regime of an Einstein-de Sitter universe, the growth factor is proportional to the scale factor, which leads to the simple relationship $\frac{dD}{dt} = H(t)D(t)$. This allows us to rewrite the initial velocity as:

$$\mathbf{v}_{\text{pec}}(t_{initial}) = H(t_{initial})D(t_{initial})\mathbf{\Psi}_0(\mathbf{x}_{\text{grid}})$$

This method produces a self-consistent set of initial conditions for both position and velocity, where the particle motions are correlated over large distances, forming the beginnings of the filaments and voids that will later evolve into galaxies and clusters.

Reference Sirko, E. (2005). *Initial Conditions to Cosmological N-Body Simulations, or Translations from the Power Spectrum to Real Space.* arXiv:astro-ph/0503106. Available at https://arxiv.org/abs/astro-ph/0503106

Validation and Accuracy

Conservation of Energy and Momentum

A physically meaningful simulation must obey the same fundamental laws as the universe it models. For a closed system governed by gravity, the two most powerful checks are therefore the laws of conservation of energy and momentum. If a simulation violates these "golden rules", it is a clear sign of a fundamental error in its implementation or underlying model.

It is crucial to understand that the rules of **conservation of energy and momentum** are only valid for a closed, non-expanding system. The total energy of a system of particles is *not* a conserved quantity in an expanding universe. The expansion of space itself does work on the system. The peculiar velocities of particles decrease due to Hubble drag, and the potential energy changes as the physical distances between all particles grow. Therefore, checking for energy conservation during a cosmological run is not a valid test; the energy is expected to change over time.

To use conservation as a test of a simulation's accuracy, we must first disable the cosmic expansion. This is done by setting the scale factor to a constant value, $a(t) = 1$, for the entire run. In this "static box" mode, the universe does not expand, and the Hubble drag term is zero. The simulation becomes a pure gravitational N-body problem.

The standard practice is to first validate the code in a non-expanding box to confirm these conservation laws hold to a high degree. Once the core engine is verified, we can then enable the cosmic expansion, confident that any subsequent physical behavior is a result of the cosmology, not a bug in the integrator.

Conservation of Momentum

In a closed system with no external forces, the total momentum must remain constant. The conservation of momentum is a direct consequence of Newton's third law: for every force $\mathbf{F}_{ij}$ that particle j exerts on particle i, there is an equal and opposite force $\mathbf{F}_{ji} = -\mathbf{F}_{ij}$. When we sum the forces over the entire system, all these internal forces perfectly cancel out. If the code correctly implements this symmetry, total momentum will be conserved.

The total momentum $\mathbf{P} = (P_x, P_y, P_z)$ can be computed as:

$$P_x = \sum_i m_i v_{ix}$$

$$P_y = \sum_i m_i v_{iy}$$

$$P_z = \sum_i m_i v_{iz}$$

Where m_i is the mass of particle i, and v_{ix}, v_{iy} and v_{iz} are its velocity components. The values of P_x, P_y and P_z should remain unchanged along the simulation to within machine precision. Any systematic drift indicates a bug in the force calculation.

In a closed system with no external torques, the total **angular momentum** must also be conserved. For a system of particles, the total angular momentum is the vector sum of each particle's individual angular momentum, $\mathbf{L} = \mathbf{r} \times \mathbf{p}$.

We can calculate the total angular momentum vector $\mathbf{L} = (L_x, L_y, L_z)$ using the formula:

$$L_x = \sum_i m_i (y_i v_{iz} - z_i v_{iy})$$

$$L_y = \sum_i m_i (z_i v_{ix} - x_i v_{iz})$$

$$L_z = \sum_i m_i (x_i v_{iy} - y_i v_{ix})$$

Just like with linear momentum, the values of L_x, L_y, and L_z should each remain unchanged. Any systematic drift in any component signals a subtle bug in the geometric implementation of your force.

Conservation of Energy

For a conservative system like gravity, the total energy—the sum of the **Kinetic Energy (KE)** from motion and the **Potential Energy (PE)** from gravitational attraction—must remain constant.

This is a much more sensitive and comprehensive test than momentum conservation. It validates the entire simulation loop, especially the accuracy of the **time integrator**. While momentum checks the symmetry of the forces at a single instant, energy conservation checks how well the simulation evolves the system from one state to the next.

Because the simulation moves in discrete time steps, we don't expect the energy to be conserved perfectly. Instead, the *behavior* of the error tells us if the integrator is working correctly:

- **A good (symplectic) integrator** will produce an energy error that **oscillates** around the initial value. The energy will wobble, but it will not show a long-term trend of increasing or decreasing.
- **A bad or inconsistent integrator** will cause the energy to **drift** steadily over time, indicating a systematic error that is continuously adding or removing energy from the system.

To calculate the total energy, $E(0) = KE + PE$ we just add the kinetic and potential energies as described below.

- **Kinetic Energy (KE):** The total kinetic energy is the sum of the kinetic energy of every particle in the system.

  $$KE = \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2$$

  Where m_i is the mass of particle i and v_i is its speed ($v_i^2 = v_{ix}^2 + v_{iy}^2 + v_{iz}^2$).

- **Potential Energy (PE):** The total potential energy is the sum of the potential energy for every *unique pair* of particles. The most meaningful way to measure the simulation's accuracy is to check how well it conserves the total energy of the **ideal, unsoftened system**.

  $$PE = \sum_{i=1}^{N} \sum_{j>i}^{N} \frac{-G m_i m_j}{r_{ij}}$$

  Where r_{ij} is the distance between particles i and j.

We can periodically recalculate this total energy, $E(t)$, as the simulation runs and monitor the relative error over time: $\frac{E(t)-E(0)}{E(0)}$.

A small, bounded oscillation in this value is the signature of a healthy, stable simulation. A consistent drift, no matter how small, points to a deeper issue that needs to be fixed.

The Two-Body Problem and Kepler's Laws

The conservation laws of energy and momentum are powerful checks on the overall stability of our simulation, but they don't tell us if the individual trajectories of our particles are physically correct. For that, we need a "ground truth"—a simple problem with a known, exact mathematical solution that we can compare our simulation against.

In celestial mechanics, the perfect "unit test" is the **two-body problem**. It is one of the very few gravitational problems that can be solved perfectly with pen and paper, and its solution is described by **Kepler's Laws of Planetary Motion**. If a simulation can't reproduce this fundamental result, it can't be trusted with more complex systems.

The setup requires to simplify the simulation to just two bodies:

1. **The "Star":** Create one particle with a very large mass and place it near the center of the simulation box. Give it zero initial velocity to keep it mostly stationary.
2. **The "Planet":** Create a second particle with a much smaller mass and place it some distance away from the star, for example, along the x-axis. Give the planet an initial velocity purely in the y-direction. The magnitude of this velocity is crucial; a specific value will produce a circular orbit, while slightly different values will produce stable elliptical orbits.

After running the simulation, we can check the planet's trajectory against Kepler's predictions:

1. Is the Orbit a Closed Ellipse? (Kepler's First Law) The most important check. The planet should trace a stable, closed ellipse with the star at one of the foci. Common failure modes are: * **Energy Drift:** The

orbit spirals inwards or outwards, indicating a non-symplectic integrator or a bug. * **Unphysical Precession:** The ellipse itself rotates over time. A large, rapid precession is a sign of numerical inaccuracy. A stable, non-precessing ellipse is a sign of a healthy integrator.

2. Does it Speed Up and Slow Down Correctly? (Kepler's Second Law) This law states that a planet sweeps out equal areas in equal times, which is a consequence of angular momentum conservation. This means the planet must move **fastest** when it is closest to the star (perihelion) and **slowest** when it is farthest away (aphelion).

3. Does it Obey the Law of Periods? (Kepler's Third Law) The law states that the square of the orbital period (P) is proportional to the cube of the orbit's semi-major axis (a), or $P^2 \propto a^3$. The time it takes the simulated planet to complete one full orbit and the size of its orbit must satisfy this mathematical relationship.

Passing the two-body test is a critical milestone. It builds confidence that the implementations of the gravitational force and the time integrator are fundamentally sound, and it's a prerequisite for tackling the chaotic dance of a true N-body system.

Sources of Error

A perfect simulation is impossible. The goal is not to eliminate all errors but to understand where they come from, control them, and ensure they are small enough that the results are physically meaningful. In a P^3M simulation, the errors arise from the three fundamental approximations we make to turn the smooth, continuous problem of gravity into a discrete problem a computer can solve.

Time Discretization Error

Physics happens continuously, but a simulation must leap forward in discrete chunks of time, Δt. The time integrator works by assuming that the acceleration changes in a simple, predictable way during that small leap.

However, the true acceleration, especially during a close encounter, can be more complex. The difference between the integrator's assumed path and the true physical path is called **truncation error**. This error scales with the square of the time step ($O(\Delta t^2)$) and is a primary contributor to the oscillations in the total energy.

Softening Bias

To prevent infinite forces when particles get too close, we modify the force law with a softening parameter, ϵ.

$$F = \frac{Gm_1m_2}{r^2 + \epsilon^2}$$

This is not a numerical error but a **physical modeling error**. We are knowingly simulating a slightly different, "softer" version of gravity. This error is most significant at very short distances ($r \lesssim \epsilon$) and prevents the formation of very "hard," dense structures. We accept this trade-off because it grants us numerical stability, but the simulation becomes blind to any physics occurring at scales smaller than the softening length.

Spatial Discretization Error

The Particle-Mesh method gains its speed by calculating long-range forces on a grid. This introduces several errors related to the grid's finite resolution.

- **Aliasing:** The grid can only represent waves with a wavelength larger than about two cell sizes. When two particles get closer than this, their sharp, high-frequency density spike is misinterpreted by the grid as a smooth, low-frequency wave. This is **aliasing**, and it is the primary source of inaccuracy in the PM force, making it "blurry" and even repulsive at short distances.
- **Interpolation Error:** The schemes used to assign mass to the grid and interpolate forces back (like NGP or CIC) are approximations that contribute to the total error.
- **Finite Difference Error:** The force on the grid is calculated using a finite difference approximation of the gradient. This is not a perfect derivative and adds a small amount of error.

Ultimately, running a successful simulation implies balancing these interconnected errors. A smaller softening length demands a smaller time step. A finer grid reduces aliasing but increases computational cost. Understanding these trade-offs is the key to producing results that are both stable and physically reliable.

Reference Dehnen, W., & Read, J. I. (2011). *N-body simulations of gravitational dynamics.* arXiv:1105.1082. Available at https://arxiv.org/abs/1105.1082.