BAO Reconstruction Evolution Report

Table of Contents

Executive Summary

Here is an executive summary of the BAO reconstruction algorithm evolution experiment:

Executive Summary: Evolutionary Optimization of BAO Reconstruction

This experiment successfully demonstrated the power of evolutionary algorithms to enhance Baryon Acoustic Oscillation (BAO) reconstruction, achieving a substantial 22.8% improvement in reconstruction quality over standard baselines. Over the course of 721 hours and 1,165 generations, the system evolved a highly effective algorithm that raised the mean cross-correlation coefficient ($r(k)$) in the critical BAO range ($k \in [0.01, 0.5]$ h/Mpc) from 0.7526 to 0.9239. Most notably, the evolved solution dramatically improved performance at smaller scales (higher k), where traditional methods typically falter due to non-linear structure formation. For instance, at $k=0.1881$ h/Mpc, the correlation improved by nearly 3.5%, significantly extending the range of reliable cosmological data extraction.

The primary trade-off for this precision is computational cost, with the best-performing algorithm requiring approximately 76 seconds per realization compared to the baseline's 9.7 seconds. However, this 7x increase in runtime yields a scientifically critical gain in signal fidelity. The evolved approach maintained near-perfect large-scale correlations ($r > 0.995$ for $k < 0.2$) while boosting the overall mean correlation of the density field from 0.345 to 0.580. This suggests the algorithm has learned to better model non-linear displacements rather than simply smoothing the density field, effectively

reversing gravitational evolution more accurately than standard Zeldovich approximation-based methods.

For cosmological analysis, these results imply a potential reduction in statistical errors for distance measurements derived from galaxy surveys. By recovering information at higher wavenumbers ($k > 0.15$ h/Mpc), the evolved algorithm effectively increases the useful volume of survey data. The ability to push reconstruction validity deeper into the non-linear regime offers a pathway to tighter constraints on Dark Energy parameters and the expansion history of the universe without requiring additional observational time. While the increased computational demand is non-negligible, it remains well within the feasible range for modern high-performance computing pipelines used in major surveys like DESI or Euclid.

Evolution Overview

Metric	Value
Duration	721.46 hours
Total Generations	1165
Programs Evaluated	1168
Successful Programs	901 (77.1%)
Best Found at Generation	1144
Initial Score	0.7526
Final Best Score	0.9239
Relative Improvement	22.8%

Evolution Progress

```
Generation | Best Score
-----------|-----------
         0 | 0.752577
       106 | 0.769830
       172 | 0.818503
       246 | 0.842625
       312 | 0.846708
```

```
 379 | 0.859223
 442 | 0.868764
 508 | 0.871521
 571 | 0.879741
 634 | 0.216742
 701 | 0.882500
 775 | 0.876910
 845 | 0.884823
 936 | 0.883987
1028 | 0.922489
1137 | 0.874586
```

Reconstruction Quality Metrics

Metric	Baseline	Best	Improvement
Combined Score	0.7526	0.9239	+0.1713
Mean r(k) BAO Range [0.01, 0.5]	0.7526	0.9239	+0.1713 (+22.8%)
Mean r(k) Large Scale [0.01, 0.2]	0.9881	0.9957	+0.0075
Mean Correlation	0.3450	0.5797	+0.2347
Max Degradation	0.0000	0.0000	+0.0000
Penalty Applied	0.0000	0.0000	+0.0000
Computation Time (s)	9.65	76.09	+66.44

Per-k Large-Scale r(k) Values

k (h/Mpc)	Baseline r(k)	Best r(k)	Reference	Improvement
0.0219	0.9995	0.9996	0.9994	+0.0001
0.0456	0.9984	0.9991	0.9983	+0.0007
0.0694	0.9973	0.9985	0.9972	+0.0012
0.0931	0.9964	0.9977	0.9964	+0.0013

k (h/Mpc)	Baseline r(k)	Best r(k)	Reference	Improvement
0.1169	0.9950	0.9965	0.9947	+0.0016
0.1406	0.9899	0.9951	0.9882	+0.0052
0.1644	0.9759	0.9916	0.9716	+0.0158
0.1881	0.9526	0.9870	0.9459	+0.0345

Tunable Parameters

The best algorithm uses the following tunable parameters, optimized via automatic differentiation:

Parameter	Value	Bounds	Method
`bias`	2.315250	(0.80, 3.00)	autodiff
`init_R_perp`	7.354290	(1.00, 15.00)	autodiff
`init_R_para`	5.571760	(1.00, 15.00)	autodiff
`gamma_adv`	0.961606	(0.00, 3.00)	autodiff
`gamma_I2`	-3.550830	(-5.00, 1.00)	autodiff
`mix_coeff`	2.420240	(0.00, 4.00)	autodiff
`hk_strength`	5.345250	(-6.00, 6.00)	autodiff
`c0_threshold`	0.560210	(0.05, 0.70)	autodiff
`c0_slope`	-1.437450	(-2.00, 2.00)	autodiff
`dewarp_ridge_pow`	-2.115290	(-6.00, -0.50)	autodiff

Parameter Descriptions

- `bias`: Galaxy bias parameter relating observed galaxy density to underlying matter density. Higher values indicate stronger bias.
- `init_R_perp`: Initial smoothing scale (Mpc/h) perpendicular to the line of sight. Controls transverse smoothing of the density field.

- `init_R_para` : Initial smoothing scale (Mpc/h) parallel to the line of sight. Controls radial smoothing, accounting for redshift-space distortions.
- `gamma_adv` : Advection correction coefficient. Controls the strength of the advective term s·∇δ in the reconstruction.
- `gamma_I2` : Second invariant coefficient. Controls contribution of I_2 (related to shear) to the source term.
- `mix_coeff` : Mixing coefficient for blending different displacement field channels (primary Zeldovich + corrections).
- `hk_strength` : High-k regularization strength. Controls suppression of small-scale (high wavenumber) modes to prevent noise amplification.
- `c0_threshold` : Threshold parameter for adaptive correction. Determines where correction kicks in based on local density.
- `c0_slope` : Slope parameter controlling the rate of correction transition around the threshold.
- `dewarp_ridge_pow` : Power-law exponent for dewarp ridge regularization. Controls scale-dependent damping in iterative steps.

Baseline Algorithm Analysis

Scientific Interpretation of the Baseline BAO Reconstruction Algorithm

Core Approach The algorithm implements standard **Zeldovich Reconstruction**, a first-order Lagrangian perturbation theory approach widely used in cosmology. It operates by estimating the displacement field (Ψ) from the observed density field using the Zeldovich approximation, where $\Psi(\mathbf{k}) \propto -i\mathbf{k}/k^2 \delta(\mathbf{k})$. The method attempts to reverse the bulk flows caused by gravity by moving mass elements back toward their initial Lagrangian positions. This specific implementation is a "paint-shift-paint" scheme, where the density field is treated as a collection of particles on a grid, shifted by the calculated displacement, and then repainted to form the reconstructed field.

Key Features and Physical Meaning The reconstruction process follows three distinct physical steps: 1. **Displacement Estimation:** The code performs an FFT on the overdensity field and applies a Gaussian smoothing kernel (`smoothing_scale_R_s`). This smoothing is physically critical; it suppresses small-scale non-linearities (high k) where the Zeldovich approximation breaks down, ensuring the displacement field is derived only from the linear, large-scale modes. 2. **Particle Shifting:** It generates a uniform grid of "particles" and shifts them by the calculated displacement vector. This effectively reverses the gravitational collapse. 3. **Differential Density Calculation:** The final reconstructed field is derived using the difference between a shifted data field (`delta_d`) and a shifted random field (`delta_s` , approximated here by shifting a uniform grid of ones). This differential approach, $(\delta_d - \delta_s) / b$, is standard practice to remove the purely geometric distortions introduced by the shifting process itself, isolating the recovered initial density signal.

Novel Elements While the physics is standard, the implementation utilizes JAX (`jnp`) for high-performance computing. The use of `jnp.meshgrid` and vectorized operations in `Shift` and `CICPaint_Single` (Cloud-in-Cell) indicates a focus on differentiability and GPU/TPU acceleration. The `CICPaint_Single` function uses `grid.at[...].add(...)` , which is JAX's specific syntax for in-place updates, crucial for handling the "scatter" operation where multiple particles contribute to the same grid cell. The algorithm also exposes the smoothing scale and bias as tunable hyperparameters for an evolutionary optimizer, allowing the reconstruction to adapt to specific simulation characteristics automatically.

Strengths and Limitations * **Strengths:** The algorithm is computationally efficient (O(N log N) due to FFTs) and mathematically robust for the linear regime. The inclusion of the shifted random field term (`delta_s`) ensures that the reconstruction does not introduce artificial mode coupling due to the grid distortions. The JAX implementation makes it highly parallelizable and differentiable. * **Limitations:** As a first-order method, it does not account for higher-order Lagrangian perturbation terms (2LPT), limiting its accuracy in recovering information from the quasi-linear regime. Furthermore, the "randoms" are approximated by a uniform grid of ones rather than a true catalog of random particles with the same survey geometry (mask/selection function), which assumes a periodic box simulation context rather than observational survey data.

Best Evolved Algorithm Analysis

Scientific Interpretation of the Evolved BAO Reconstruction Algorithm

Core Approach: This algorithm employs a **hybrid iterative displacement field reconstruction** method, rooted in the Zeldovich approximation but significantly augmented by higher-order Lagrangian perturbation theory (LPT) terms and a multi-stage spectral filtering process. It operates in two distinct phases: a primary bulk-flow removal step (standard reconstruction) followed by a sophisticated "physics augmentation" phase that attempts to recover information lost to non-linear evolution using a basis of tidal and density invariants.

Key Features: The reconstruction begins with a standard Zeldovich step, solving for the displacement field Ψ using a smoothed density field and removing redshift-space distortions (RSD) via an anisotropic kernel (`solve_disp`). The innovation lies in the subsequent steps: 1. **Tensor Invariant Augmentation:** The algorithm computes the shear tensor invariants (I_1, I_2, I_3) of the initial displacement field and uses them, along with an advection term ($\mathbf{\Psi} \cdot \nabla \delta$), to construct a secondary displacement source. This mimics 2LPT (Second-order Lagrangian Perturbation Theory) corrections but with free, evolved coefficients (`gamma_adv` , `gamma_I2` , etc.) rather than fixed theoretical values. 2. **Spectral De-Warping (Stage A):** A "Tri-Basis De-Warping" step attempts to fit the high-k residual density field using a basis set composed of δ^2, $|\nabla \delta|^2$, and tidal scalar s^2. This acts as a bias renormalization scheme, correcting for non-linear mode coupling at smaller scales. 3. **Coherence-Gated Fusion (Stage B):** The final density field is constructed by fusing the de-warped field with a set of tracers (log-density, non-linear density, scale-dependent bias

terms). This fusion is governed by a spectral coherence gate, which only allows the mixing of terms that show strong cross-correlation with the reference field in specific k-bands.

Novel Elements: The most distinct departure from standard reconstruction is the **differentiable, band-wise spectral fusion**. Standard methods usually apply a global smoothing kernel. This algorithm, however, uses `jax.lax.scan` to iterate through specific wavenumber bands ($k \sim 0.2 - 0.6$ h/Mpc), applying a Wiener-like filter that adaptively blends the reconstructed field with non-linear tracers based on their local spectral coherence. Additionally, the explicit inclusion of a "Gram-Schmidt" orthogonalization step between the primary and secondary displacement channels ensures that the higher-order corrections do not re-introduce large-scale bulk flows already handled by the Zeldovich step.

Strengths and Limitations: * **Strengths:** The algorithm is highly robust against small-scale noise due to the coherence gating mechanism. By explicitly modeling non-linear terms (tidal forces, advection), it likely recovers BAO signal in the "gray zone" ($k \approx 0.2-0.5$ h/Mpc) better than standard smoothing. The "Hard Low-k Lock" ensures that the linear regime ($k < 0.2$), which is already well-measured, is perfectly preserved, preventing the reconstruction from degrading large-scale information. * **Limitations:** The computational complexity is significantly higher than standard reconstruction (76s vs typical <10s) due to the multiple FFTs required for tensor invariants and the band-wise scanning loops. Furthermore, the reliance on many tunable "evolved" parameters (e.g., `dewarp_ridge_pow`, `c0_threshold`) implies the model might be over-fitted to the specific cosmology or simulation suite used during the evolutionary optimization, potentially requiring re-tuning for different survey specifications.

Evolution Improvements

Here is a detailed analysis of the improvements made by the evolved BAO reconstruction algorithm compared to the baseline.

1. Key Modifications

The baseline algorithm is a standard implementation of **Zeldovich Reconstruction**, which smooths the density field, computes a displacement field using the linear Zeldovich approximation, and shifts particles back.

The evolved algorithm transforms this into a **Multi-Stage, Non-Linear Reconstruction Pipeline**. The key modifications are:

- **Anisotropic Smoothing:** Instead of a single isotropic Gaussian smoothing scale (R_s), the evolved code uses separate perpendicular ($R_\perp \approx 7.35$) and parallel ($R_\parallel \approx 5.57$) smoothing scales.
- **Physics-Augmented Displacement:** It introduces a second pass for the displacement field. It calculates "invariants" of the deformation tensor (shear terms I_1, I_2, I_3) and an advection

term, mixing them into the displacement calculation.
- **Iterative "De-Warping" (Stage A):** A post-reconstruction step that attempts to clean up the reconstructed field by regressing it against non-linear basis functions (density squared, gradients, tidal scalars) in specific k-bands.
- **Tracer Fusion (Stage B):** A complex weighting scheme that blends the reconstructed field with various "tracers" (log-density, non-linear density, scale-dependent bias terms) based on their coherence with the signal in Fourier space.
- **High-k Refinement:** Explicit logic to treat high-k (small scale) modes differently, applying aggressive filtering and substitution where the reconstruction is known to be noisy.

2. Physical Justification

Does it make physical sense? Mostly **Yes**, though with high complexity.

- **Anisotropic Smoothing:** This is physically well-motivated. Redshift Space Distortions (RSD) elongate structures along the line of sight (the "Fingers of God" effect) and squash them on large scales (Kaiser effect). Using different smoothing scales for parallel vs. perpendicular directions accounts for this asymmetry, allowing the algorithm to recover information more effectively in the presence of RSD.
- **Shear Invariants (I_2, I_3):** Standard reconstruction assumes the displacement is purely the gradient of a potential (irrotational). However, non-linear structure formation generates tidal forces and shear. Including the invariants of the shear tensor (I_2 specifically relates to tidal torque) attempts to correct for higher-order Lagrangian perturbation theory (2LPT) effects that the standard Zeldovich approximation misses.
- **Advection Term:** The term `gamma_adv * adv` represents the transport of density anomalies by the bulk flow. Standard reconstruction moves particles; this term attempts to model the *change* in the density contrast caused by that movement mathematically, effectively acting as a higher-order correction to the continuity equation.
- **Scale-Dependent Bias:** The inclusion of `delta_sdbias_k` acknowledges that galaxies (tracers) do not trace dark matter perfectly linearly at all scales.

Critique: The "De-Warping" and "Fusion" stages act more like **denoising filters** or **Wiener filters** than pure physical reconstruction. They rely on the correlation between non-linear terms and the noise to subtract out mode-coupling effects. While mathematically effective for maximizing $r(k)$, they move away from the pure dynamical restoration of particle trajectories.

3. Novel Techniques

Several innovative approaches appear in the evolved code:

- **Spectral "Band-Scanning" (Jax `scan`):** The algorithm processes the power spectrum in distinct bands (using `_dewarp_scan` and `_fusion_scan`). Instead of a global fit, it optimizes the cleaning coefficients locally in k-space. This is highly sophisticated and allows the algorithm to adapt to how noise properties change from large to small scales.

- **Coherence-Gated Mixing:** The algorithm calculates the "coherence" (correlation coefficient) between the reconstructed field and its various non-linear tracers on the fly. It uses `sigmoid` gates to only allow mixing when the coherence is high. This prevents the algorithm from injecting noise in regions where the non-linear models are poor predictors.
- **Gram-Schmidt Orthogonalization of Displacements:** The code orthogonalizes the secondary displacement field (s_2) against the primary one (s_1). This ensures that the "physics augmentation" adds *new* information rather than just rescaling the linear Zeldovich term.

4. Trade-offs

- **Computation Time vs. Accuracy:**
  - **Baseline:** 9.65s
  - **Evolved:** 76.09s (+66.44s)
  - The evolved algorithm is nearly **8x slower**. This is a significant cost. However, for cosmological surveys where data processing is done once, this cost is likely acceptable given the massive gain in signal fidelity.
- **Complexity vs. Maintainability:** The evolved code is drastically more complex. It moves from a simple FFT-shift-IFFT procedure to a multi-stage pipeline involving tensor calculus, iterative scanning, and dozens of tunable hyperparameters. Debugging this system or deriving analytic covariance matrices for it would be extremely difficult.
- **Linearity vs. Non-Linearity:** The baseline preserves the linear relationship between input and output relatively well. The evolved algorithm introduces highly non-linear gating and thresholding. This could potentially introduce subtle biases in the power spectrum amplitude or quadrupole that would need careful calibration.

5. Scientific Validity and Generalization

Is it overfit? There is a **moderate risk of overfitting**. The algorithm has introduced a large number of "magic numbers" (e.g., `c0_threshold`, `mix_coeff`, `dewarp_ridge_pow`, specific band centers `[0.23, 0.30...]`). * If these parameters were tuned on a specific simulation box at a specific redshift, they might not perform as well on a survey with different shot noise levels, galaxy bias, or redshift. * However, the *structure* of the solution (using shear invariants and anisotropic smoothing) is robust.

The r(k) Improvement: The improvement in $r(k)$ at high k (0.188 h/Mpc) is distinct: **0.9526 $\to$ 0.9870**. This is physically significant. In BAO analysis, the "damping" of the acoustic peak is dominated by non-linear bulk flows. By recovering the correlation to nearly 0.99 up to $k=0.2$, the algorithm has effectively undone almost all non-linear smearing relevant for BAO cosmology. This would translate directly to tighter constraints on Dark Energy parameters (w_0, w_a).

Conclusion: The evolved algorithm represents a shift from "Dynamical Reconstruction" to "Hybrid Dynamical-Statistical Reconstruction." It uses dynamics to move the mass, but then uses statistical learning techniques (in the Fourier domain) to clean up the resulting field. It is a highly effective, albeit computationally expensive, "super-resolution" technique for density fields.

Experiment Configuration

Setting	Value
Number of Islands	5
Migration Interval	10 generations
Max Generations	10000
LLM Models Used	gemini-3-pro-preview, gpt-5.2, gemini-3-flash-preview, o4-mini

Task Description

The algorithm evolves to solve BAO (Baryon Acoustic Oscillation) reconstruction: - Recover the initial density field from evolved observations - Maximize cross-correlation r(k) in BAO range (k ~ 0.01-0.5 h/Mpc) - Preserve large-scale structure without degradation - Key constraint: r(k) at large scales must not degrade from baseline

Appendix: Algorithm Code

Baseline Algorithm

```
# EVOLVE-BLOCK-START

# TUNABLE: smoothing_scale_R_s = 9.08926, bounds=(1.0, 30.0), method=autodiff
# TUNABLE: bias = 1.02773, bounds=(1.0, 3.0), method=autodiff

def reconstruct(data: jnp.ndarray, smoothing_scale_R_s: float = 9.08926,
                bias: float = 1.02773, box_size: float = 1000.0) -> jnp.ndarray:
    """
    Performs 3D BAO reconstruction for dark matter density fields.

    Args:
        data: Input 3D density field (dark matter overdensity) as a NumPy array.
              The input is expected to be a cubic array.
        smoothing_scale_R_s: Gaussian smoothing scale in Mpc/h.
                             The k-space density field delta_k is multiplied by
                             exp(-k^2 * R_s^2 / 2) where k includes the 2π factor.
                             This suppresses small-scale noise in the displacement field.
```

```
                                If R_s <= 0, no smoothing is applied. Default is 10.0.
            bias: Bias factor for dark matter. Default is 1.02773.
            box_size: Box size in Mpc/h. Default is 1000.0.

        Returns:
            Reconstructed 3D density field as a NumPy array.
                  The output array will have the same dtype as the input array.
        """
        if not isinstance(data, jnp.ndarray):
            raise TypeError("Input data must be a NumPy array.")
        if data.ndim != 3:
            raise ValueError("Input data must be a 3D array.")
        if not (data.shape[0] == data.shape[1] == data.shape[2]):
            raise ValueError("Input data must be a cubic 3D array.")

        N = data.shape[0]

        if N == 0:
            return jnp.array([], dtype=data.dtype).reshape(0, 0, 0)

        reconstructed_field = apply_reconstruction(data, smoothing_scale_R_s, bias, box_size)

        return reconstructed_field


    def apply_reconstruction(density_field, smoothing_scale, bias, BoxSize):
        """Applies standard reconstruction to a density field.

        Standard reconstruction in cosmology is a technique used to sharpen the
        baryon acoustic oscillation (BAO) signal by removing non-linear effects
        of structure formation. It works by estimating the displacement field
        from the observed density field and then moving galaxies back to their
        initial positions.

        Args:
            density_field: 3D numpy array representing the density field
            smoothing_scale: smoothing scale in Mpc/h for the Gaussian filter
            bias: Bias factor for dark matter
            BoxSize: Box size in Mpc/h

        Returns:
            reconstructed_field: 3D numpy array of the reconstructed density field
        """
        # Get dimensions of the density field
        nx, ny, nz = density_field.shape

        # Convert density field to overdensity field
        overdensity = density_field - 1.0

        # FFT of the overdensity field
        delta_k = jnp.fft.fftn(overdensity)

        # Create k-space grid
        kx = jnp.fft.fftfreq(nx, 1/nx) * 2 * jnp.pi / BoxSize
        ky = jnp.fft.fftfreq(ny, 1/ny) * 2 * jnp.pi / BoxSize
        kz = jnp.fft.fftfreq(nz, 1/nz) * 2 * jnp.pi / BoxSize

        kx_grid, ky_grid, kz_grid = jnp.meshgrid(kx, ky, kz, indexing='ij')
```

```
        k_sq = kx_grid**2 + ky_grid**2 + kz_grid**2

        # Apply Gaussian smoothing
        smoothing_factor = jnp.exp(-0.5 * k_sq * smoothing_scale**2)

        # Avoid division by zero
        k_sq = k_sq.at[0, 0, 0].set(1.0)

        # Calculate displacement field in k-space
        # Using the Zeldovich approximation: Ψ(k) = -ik/k² δ(k)
        displacement_x_k = -1j * kx_grid / k_sq * delta_k * smoothing_factor / bias
        displacement_y_k = -1j * ky_grid / k_sq * delta_k * smoothing_factor / bias
        displacement_z_k = -1j * kz_grid / k_sq * delta_k * smoothing_factor / bias

        # Transform back to real space
        displacement_x = jnp.real(jnp.fft.ifftn(displacement_x_k))
        displacement_y = jnp.real(jnp.fft.ifftn(displacement_y_k))
        displacement_z = jnp.real(jnp.fft.ifftn(displacement_z_k))

        # Apply displacement to get reconstructed field
        reconstructed_field = apply_displacement(density_field.astype(jnp.float32),
                                                 displacement_x.astype(jnp.float32),
                                                 displacement_y.astype(jnp.float32),
                                                 displacement_z.astype(jnp.float32),
                                                 BoxSize,
                                                 bias)

        return reconstructed_field


    def apply_displacement(density_field, disp_x, disp_y, disp_z, BoxSize, bias):
        """Applies displacement field to the density field.

        Args:
            density_field: Original density field
            disp_x, disp_y, disp_z: Components of the displacement field
            BoxSize: Box size in Mpc/h
            bias: Bias factor for dark matter

        Returns:
            Reconstructed density field
        """
        nx, ny, nz = density_field.shape

        # Create grid particles
        H = BoxSize/nx
        X = jnp.arange(nx)*H
        Y = jnp.arange(ny)*H
        Z = jnp.arange(nz)*H
        Particle_grid = (jnp.array(jnp.meshgrid(X, Y, Z, indexing='ij')).reshape([3,
    -1]).transpose()).astype(jnp.float32)

        # Shift particles
        Position_shifted = Shift(Particle_grid, disp_x, disp_y, disp_z, nx, BoxSize, nx * ny * nz)

        # Paint shifted particles onto grid using CIC implementation
        delta_d = CICPaint_Single(Position_shifted, density_field.reshape(-1), nx, BoxSize, nx * ny *
    nz) - 1
```

```
    delta_s = CICPaint_Single(Position_shifted, jnp.ones(nx * ny * nz, dtype=jnp.float32), nx,
BoxSize, nx * ny * nz) - 1

    # Apply shift
    reconstructed_field = (delta_d - delta_s) / bias

    return reconstructed_field


def Shift(particle_positions, disp_x, disp_y, disp_z, nmesh, box_size, num_particles=None):
    """
    Optimized vectorized implementation of particle position shifting with displacement fields.

    Args:
        particle_positions: Array of particle positions, shape (num_particles, 3)
        disp_x, disp_y, disp_z: 3D displacement field arrays, shape (nmesh, nmesh, nmesh)
        nmesh: Grid size
        box_size: Box size in physical units
        num_particles: Number of particles

    Returns:
        shifted_positions: Array of shifted particle positions with periodic boundary conditions
    """
    # Grid spacing
    cell_size = box_size / nmesh

    # Get particle positions
    pos_x = particle_positions[:, 0]
    pos_y = particle_positions[:, 1]
    pos_z = particle_positions[:, 2]

    # Find grid cell indices for all particles at once (vectorized)
    ix = (pos_x / cell_size).astype(int) % nmesh
    iy = (pos_y / cell_size).astype(int) % nmesh
    iz = (pos_z / cell_size).astype(int) % nmesh

    # Get displacements at particle positions using advanced indexing
    disp_at_x = disp_x[ix, iy, iz]
    disp_at_y = disp_y[ix, iy, iz]
    disp_at_z = disp_z[ix, iy, iz]

    # Apply displacement to all particles at once
    new_x = pos_x + disp_at_x
    new_y = pos_y + disp_at_y
    new_z = pos_z + disp_at_z

    # Apply periodic boundary conditions (vectorized)
    new_x = jnp.mod(new_x, box_size)
    new_y = jnp.mod(new_y, box_size)
    new_z = jnp.mod(new_z, box_size)

    # Stack results
    shifted_positions = jnp.column_stack([new_x, new_y, new_z])

    return shifted_positions.astype(jnp.float32)


def CICPaint_Single(particle_positions, particle_values, nmesh, box_size, num_particles=None):
```

```
    """
    Optimized vectorized implementation of Cloud-In-Cell (CIC) painting.
    Assigns particle values to a grid using CIC interpolation.

    Args:
        particle_positions: Array of particle positions, shape (num_particles, 3)
        particle_values: Array of particle values, shape (num_particles,)
        nmesh: Grid size
        box_size: Box size in physical units
        num_particles: Number of particles

    Returns:
        grid: 3D grid with painted values, shape (nmesh, nmesh, nmesh)
    """
    # Initialize grid
    grid = jnp.zeros((nmesh, nmesh, nmesh), dtype=jnp.float32)

    # Grid spacing
    cell_size = box_size / nmesh

    # Get all particle positions at once
    pos_x = particle_positions[:, 0]
    pos_y = particle_positions[:, 1]
    pos_z = particle_positions[:, 2]

    # Convert to grid coordinates
    x_grid = pos_x / cell_size
    y_grid = pos_y / cell_size
    z_grid = pos_z / cell_size

    # Find lower corner indices for all particles
    ix0 = jnp.floor(x_grid).astype(int)
    iy0 = jnp.floor(y_grid).astype(int)
    iz0 = jnp.floor(z_grid).astype(int)

    # Calculate fractional distances from lower corner for all particles
    dx = x_grid - ix0
    dy = y_grid - iy0
    dz = z_grid - iz0

    # Process each of the 8 corners
    for di in range(2):
        for dj in range(2):
            for dk in range(2):
                # Calculate indices for this corner
                ix = (ix0 + di) % nmesh
                iy = (iy0 + dj) % nmesh
                iz = (iz0 + dk) % nmesh

                # Calculate CIC weights for all particles at once
                wx = dx if di == 1 else (1 - dx)
                wy = dy if dj == 1 else (1 - dy)
                wz = dz if dk == 1 else (1 - dz)
                weights = wx * wy * wz

                # Add contributions to grid using np.add.at
                # This handles multiple particles contributing to the same grid point
                grid = grid.at[ix, iy, iz].add(particle_values * weights)
```

```
    return grid

# EVOLVE-BLOCK-END
```

Best Evolved Algorithm

```
# EVOLVE-BLOCK-START

# TUNABLE: bias                  = 2.31525,    bounds=(0.8, 3.0),    method=autodiff
# TUNABLE: init_R_perp           = 7.35429,    bounds=(1.0, 15.0),   method=autodiff
# TUNABLE: init_R_para           = 5.57176,    bounds=(1.0, 15.0),   method=autodiff
# TUNABLE: gamma_adv             = 0.961606,        bounds=(0.0, 3.0),    method=autodiff
# TUNABLE: gamma_I2              = -3.55083,   bounds=(-5.0, 1.0),   method=autodiff
# TUNABLE: mix_coeff             = 2.42024,     bounds=(0.0, 4.0),    method=autodiff
# TUNABLE: hk_strength           = 5.34525,    bounds=(-6.0, 6.0),   method=autodiff
# TUNABLE: c0_threshold          = 0.56021,        bounds=(0.05, 0.7),   method=autodiff
# TUNABLE: c0_slope              = -1.43745,        bounds=(-2.0, 2.0),   method=autodiff
# TUNABLE: dewarp_ridge_pow      = -2.11529,       bounds=(-6.0, -0.5),  method=autodiff

def _fftn_batched(x):
    return jnp.fft.fftn(x, axes=(-3, -2, -1))

def _ifftn_batched(x):
    return jnp.fft.ifftn(x, axes=(-3, -2, -1))

def _compute_tri_invariants(s_x, s_y, s_z, kx_grid, ky_grid, kz_grid):
    sx_k = jnp.fft.fftn(s_x)
    sy_k = jnp.fft.fftn(s_y)
    sz_k = jnp.fft.fftn(s_z)

    grad_sx = jnp.real(_ifftn_batched(jnp.stack([1j * kx_grid * sx_k,
                                                 1j * ky_grid * sx_k,
                                                 1j * kz_grid * sx_k], axis=0)))
    grad_sy = jnp.real(_ifftn_batched(jnp.stack([1j * kx_grid * sy_k,
                                                 1j * ky_grid * sy_k,
                                                 1j * kz_grid * sy_k], axis=0)))
    grad_sz = jnp.real(_ifftn_batched(jnp.stack([1j * kx_grid * sz_k,
                                                 1j * ky_grid * sz_k,
                                                 1j * kz_grid * sz_k], axis=0)))

    T_xx, T_xy, T_xz = grad_sx[0], grad_sx[1], grad_sx[2]
    T_yx, T_yy, T_yz = grad_sy[0], grad_sy[1], grad_sy[2]
    T_zx, T_zy, T_zz = grad_sz[0], grad_sz[1], grad_sz[2]

    I1 = T_xx + T_yy + T_zz
    Tr_T_sq = (T_xx**2 + T_xy*T_yx + T_xz*T_zx +
               T_yx*T_xy + T_yy**2 + T_yz*T_zy +
               T_zx*T_xz + T_zy*T_yz + T_zz**2)
    I2 = 0.5 * (I1**2 - Tr_T_sq)
    I3 = (T_xx * (T_yy * T_zz - T_yz * T_zy) -
```

```
            T_xy * (T_yx * T_zz - T_yz * T_zx) +
            T_xz * (T_yx * T_zy - T_yy * T_zx))
        return I1, I2, I3

    def _compute_tidal_invariant(d1, kx, ky, kz, k_sq_safe):
        d1_k = jnp.fft.fftn(d1)
        H_stack_k = jnp.stack([
            -(kx**2 / k_sq_safe) * d1_k,
            -(ky**2 / k_sq_safe) * d1_k,
            -(kz**2 / k_sq_safe) * d1_k,
            -(kx*ky / k_sq_safe) * d1_k,
            -(kx*kz / k_sq_safe) * d1_k,
            -(ky*kz / k_sq_safe) * d1_k
        ], axis=0)
        H = jnp.real(_ifftn_batched(H_stack_k))
        Hxx, Hyy, Hzz = H[0], H[1], H[2]
        Hxy, Hxz, Hyz = H[3], H[4], H[5]
        sum_sq = (Hxx**2 + Hyy**2 + Hzz**2 + 2*(Hxy**2 + Hxz**2 + Hyz**2))
        s2 = sum_sq - (1.0/3.0) * (d1**2)
        return s2

    def reconstruct(
        data: jnp.ndarray,
        smoothing_scale_R_s: float = 6.96604,
        bias: float = 2.31525,
        box_size: float = 1000.0,
        init_R_perp: float = 7.35429,
        init_R_para: float = 5.57176,
        gamma_adv: float = 0.961606,
        gamma_I2: float = -3.55083,
        mix_coeff: float = 2.42024,
        hk_strength: float = 5.34525,
        c0_threshold: float = 0.56021,
        c0_slope: float = -1.43745,
        dewarp_ridge_pow: float = -2.11529,
    ) -> jnp.ndarray:
        n = data.shape[0]
        overd = data - 1.0

        # Fixed parameters
        gamma_div = 2.0
        gamma_I3 = 5.0
        stab_scale = 1.12
        guard_k = 0.01
        c_s2 = 0.12

        # Grid setup
        k1d = 2 * jnp.pi * jnp.fft.fftfreq(n, d=box_size / n)
        kx, ky, kz = jnp.meshgrid(k1d, k1d, k1d, indexing='ij')
        k_perp_sq, k_par_sq = kx**2 + ky**2, kz**2
        k_sq = k_perp_sq + k_par_sq
        k_sq_safe = jnp.where(k_sq == 0.0, 1.0, k_sq)
        k_mag = jnp.sqrt(k_sq)
        H = box_size / n
        k_nyq = jnp.pi / H

        def kernel(Rp, Rl):
            arg = jnp.sqrt(k_perp_sq * Rp**2 + 1e-24) + jnp.sqrt(k_par_sq * Rl**2 + 1e-24)
```

```
        return jnp.exp(-0.5 * arg)

    def solve_disp(delta_k, Rp, Rl, eff_bias):
        fac = (-1j) * delta_k * kernel(Rp, Rl) / (eff_bias * k_sq_safe)
        s_vec = jnp.real(_ifftn_batched(jnp.stack([kx * fac, ky * fac, kz * fac], axis=0)))
        return s_vec[0], s_vec[1], s_vec[2]

    # --- Pass 1: Standard Zeldovich ---
    overd_k = jnp.fft.fftn(overd)
    s0x, s0y, s0z = solve_disp(overd_k, init_R_perp, init_R_para, bias)
    rec0 = _apply_differentiable_displacement(data.astype(jnp.float32), s0x, s0y, s0z, box_size,
bias)

    # --- Physics Augmentation ---
    delta_res = rec0 - 1.0
    I1, I2, I3 = _compute_tri_invariants(s0x, s0y, s0z, kx, ky, kz)
    I2c = I2 - jnp.mean(I2)
    I3c = I3 - jnp.mean(I3)

    dr_k = jnp.fft.fftn(jnp.arcsinh(delta_res))
    grad_res = jnp.real(_ifftn_batched(jnp.stack([1j * kx * dr_k, 1j * ky * dr_k, 1j * kz *
dr_k], axis=0)))
    adv = s0x * grad_res[0] + s0y * grad_res[1] + s0z * grad_res[2]

    # Tunable source composition (including advection)
    src1 = (delta_res + gamma_adv * adv + gamma_div * I1 +
            gamma_I2 * I2c + gamma_I3 * I3c)

    beta = jnp.maximum(stab_scale, 1e-6)
    src1_stab = jnp.arcsinh(beta * src1) / beta
    s1x, s1y, s1z = solve_disp(jnp.fft.fftn(src1_stab), init_R_perp * 0.05, init_R_para * 0.05,
bias)

    # Secondary Channel: Tidal
    src2_stab = jnp.arcsinh(beta * 2.0 * I2c) / beta
    fac2 = 1j * jnp.fft.fftn(src2_stab) / k_sq_safe
    s2_vec = jnp.real(_ifftn_batched(jnp.stack([kx * fac2, ky * fac2, kz * fac2], axis=0)))
    s2x, s2y, s2z = s2_vec[0], s2_vec[1], s2_vec[2]

    # Gram-Schmidt Orthogonalization (s2 vs s1)
    w_hp = 1.0 / (1.0 + jnp.exp(-(k_mag - guard_k) / 0.12))
    w_hp = jnp.where(k_sq == 0.0, 0.0, w_hp)

    def apply_hp(v): return jnp.real(jnp.fft.ifftn(jnp.fft.fftn(v) * w_hp))
    s1f = [apply_hp(v) for v in (s1x, s1y, s1z)]
    # Use higher power for s2 to isolate smaller scales
    def apply_hp2(v): return jnp.real(jnp.fft.ifftn(jnp.fft.fftn(v) * w_hp**2))
    s2f = [apply_hp2(v) for v in (s2x, s2y, s2z)]

    dot_11 = jnp.mean(s1f[0]**2 + s1f[1]**2 + s1f[2]**2) + 1e-12
    dot_21 = jnp.mean(s2f[0]*s1f[0] + s2f[1]*s1f[1] + s2f[2]*s1f[2])
    proj_21 = dot_21 / dot_11

    s2o = [s2f[i] - proj_21 * s1f[i] for i in range(3)]

    mix_val = mix_coeff * (1.0 - 0.1 * jnp.tanh(I2c))
    fx = s0x + mix_val * (s1f[0] + c_s2 * s2o[0])
    fy = s0y + mix_val * (s1f[1] + c_s2 * s2o[1])
```

```
    fz = s0z + mix_val * (s1f[2] + c_s2 * s2o[2])

    # E-mode Projection & Deconvolution
    Fk = _fftn_batched(jnp.stack([fx, fy, fz], axis=0))
    div_F = kx * Fk[0] + ky * Fk[1] + kz * Fk[2]
    E_k = jnp.stack([kx * div_F / k_sq_safe, ky * div_F / k_sq_safe, kz * div_F / k_sq_safe],
axis=0)
    E_k = jnp.where(k_sq == 0.0, 0.0, E_k)

    sinc_k = (jnp.sinc(kx * H / (2*jnp.pi)) * jnp.sinc(ky * H / (2*jnp.pi)) * jnp.sinc(kz * H /
(2*jnp.pi)))**2
    taper = jnp.exp(-jnp.power(k_mag / (0.685 * k_nyq + 1e-9), 8.0))
    clean = jnp.where(k_sq == 0.0, 1.0, taper / jnp.where(sinc_k == 0.0, 1.0, sinc_k))

    disp_f = jnp.real(_ifftn_batched(E_k * clean))
    rec_final = _apply_differentiable_displacement(data.astype(jnp.float32), disp_f[0],
disp_f[1], disp_f[2], box_size, bias)

    # --- High-k Refinement ---
    delta_base_k = jnp.fft.fftn(rec_final - 1.0) * clean
    delta_base_k = jnp.where(k_sq == 0.0, 0.0, delta_base_k)

    # Windowing
    w_hk_soft = (jax.nn.sigmoid((k_mag - 0.22) / 0.02)) ** 6
    w_hi = jnp.exp(-jnp.power(k_mag / (0.85 * k_nyq + 1e-9), 6.0))
    w_hk = jnp.where(k_mag <= 0.2, 0.0, w_hk_soft * w_hi)
    w_hk = jnp.where(k_sq == 0.0, 0.0, w_hk)

    aux_filter = taper / jnp.sqrt(jnp.where(sinc_k==0, 1.0, sinc_k))

    # --- Stage A: Tri-Basis De-Warping with Tunable Ridge ---
    d1 = jnp.arcsinh(beta * (rec_final - 1.0)) / beta
    d1 = d1 - jnp.mean(d1)

    # B1: Density^2
    b1_k = jnp.fft.fftn(d1**2 - jnp.mean(d1**2)) * aux_filter

    # B2: Grad^2
    d1_k_full = jnp.fft.fftn(d1)
    g_vec = jnp.real(_ifftn_batched(jnp.stack([1j*kx*d1_k_full, 1j*ky*d1_k_full,
1j*kz*d1_k_full], axis=0)))
    g2 = jnp.sum(g_vec**2, axis=0)
    b2_k = jnp.fft.fftn(g2 - jnp.mean(g2)) * aux_filter

    # B3: Tidal Scalar
    s2_field = _compute_tidal_invariant(d1, kx, ky, kz, k_sq_safe)
    b3_k = jnp.fft.fftn(s2_field - jnp.mean(s2_field)) * aux_filter

    band_centers = jnp.array([0.23, 0.30, 0.38, 0.48, 0.62], dtype=k_mag.dtype)
    band_sigma = jnp.array(0.075, dtype=k_mag.dtype)

    def _dewarp_scan(carry, kc):
        num_acc, den_acc = carry
        wb = w_hk * jnp.exp(-0.5 * jnp.square((k_mag - kc) / (band_sigma + 1e-6)))
        wb = jnp.where(k_sq == 0.0, 0.0, wb)

        # 3x3 System Construction
        bases = [b1_k, b2_k, b3_k]
```

```
            M = jnp.zeros((3, 3))
            T = jnp.zeros(3)

            # Unrolled construction
            for i in range(3):
                T = T.at[i].set(jnp.sum(wb * jnp.real(delta_base_k * jnp.conj(bases[i]))))
                for j in range(3):
                    M = M.at[i, j].set(jnp.sum(wb * jnp.real(bases[i] * jnp.conj(bases[j]))))

            # Tunable Regularization
            ridge = (10.0**dewarp_ridge_pow) * (jnp.trace(M) + 1e-12)
            M = M + ridge * jnp.eye(3)

            coeffs = jnp.linalg.solve(M, T)
            delta_fit = coeffs[0]*b1_k + coeffs[1]*b2_k + coeffs[2]*b3_k
            return (num_acc + wb * delta_fit, den_acc + wb), None

        init_dw = (jnp.zeros_like(delta_base_k), jnp.zeros_like(k_mag))
        (num_dw, den_dw), _ = jax.lax.scan(_dewarp_scan, init_dw, band_centers)

        delta2_k = num_dw / (den_dw + 1e-12)
        delta2_k = jnp.where(k_sq == 0.0, 0.0, delta2_k)

        # Global Gating
        num_coh = jnp.sum(w_hk * jnp.real(delta_base_k * jnp.conj(delta2_k)))
        den_coh = jnp.sqrt(jnp.sum(w_hk * jnp.abs(delta_base_k)**2) * jnp.sum(w_hk *
    jnp.abs(delta2_k)**2) + 1e-12)
        coh_2 = num_coh / (den_coh + 1e-12)

        ratio = jnp.sqrt(jnp.sum(w_hk * jnp.abs(delta2_k)**2)) / (jnp.sqrt(jnp.sum(w_hk *
    jnp.abs(delta_base_k)**2)) + 1e-12)
        gate = jax.nn.sigmoid(10.0 * (coh_2 - 0.10)) * jax.nn.sigmoid(6.0 * (1.0 - ratio))

        delta_ref_k = delta_base_k - (w_hk * gate) * delta2_k
        delta_ref_k = jnp.where(k_sq == 0.0, 0.0, delta_ref_k)

        # --- Stage B: Fusion with K-Dependent Gating ---
        # Prepare tracers
        Dk = _fftn_batched(disp_f)
        delta_div_k = (-bias) * (1j * (kx * Dk[0] + ky * Dk[1] + kz * Dk[2])) * aux_filter

        logrho = jnp.log(jax.nn.softplus(rec_final) + 1e-6)
        delta_log_k = jnp.fft.fftn(logrho - jnp.mean(logrho)) * aux_filter

        dens_nl = jnp.arcsinh(beta * (rec_final - 1.0)) / beta
        delta_nl_k = jnp.fft.fftn(dens_nl - jnp.mean(dens_nl)) * aux_filter

        I3_src = jnp.arcsinh(beta * I3c) / beta
        delta_i3_k = jnp.fft.fftn(I3_src - jnp.mean(I3_src)) * aux_filter

        s0k = _fftn_batched(jnp.stack([s0x, s0y, s0z], axis=0))
        delta_div0_k = (-bias) * (1j * (kx * s0k[0] + ky * s0k[1] + kz * s0k[2])) * aux_filter

        # Scale dependent bias tracer
        bias_eff = jnp.maximum(bias * (1.0 - 0.15 * k_sq), 0.2)
        W_alt = jnp.exp(-0.5 * (jnp.power(k_perp_sq * init_R_perp**2 + 1e-24, 0.675) +
                                jnp.power(k_par_sq * init_R_para**2 + 1e-24, 0.675)))
        delta_sdbias_k = (bias / bias_eff) * overd_k * W_alt * aux_filter
```

```
    tracers = [delta_div_k, delta_log_k, delta_nl_k, delta_i3_k, delta_div0_k, delta_sdbias_k]

    a_sig = jnp.array(10.0, dtype=k_mag.dtype)

    def _fusion_scan(carry, kc):
        num_acc, den_acc, conf_acc = carry
        wb = w_hk * jnp.exp(-0.5 * jnp.square((k_mag - kc) / (band_sigma + 1e-6)))
        wb = jnp.where(k_sq == 0.0, 0.0, wb)

        # Adaptive Threshold
        c0_eff = c0_threshold + c0_slope * (kc - 0.35)

        hybrid = jnp.zeros_like(delta_ref_k)
        w_sum = jnp.zeros_like(kc)

        for t_k in tracers:
            num = jnp.sum(wb * delta_ref_k * jnp.conj(t_k))
            den = jnp.sum(wb * jnp.abs(t_k)**2) + 1e-12
            coeff = num / den
            pred = coeff * t_k

            # Real coherence for strict correlation
            num_coh = jnp.sum(wb * jnp.real(delta_ref_k * jnp.conj(pred)))
            den_coh = jnp.sqrt(jnp.sum(wb * jnp.abs(delta_ref_k)**2) * jnp.sum(wb *
jnp.abs(pred)**2) + 1e-12)
            coh = num_coh / (den_coh + 1e-12)

            w = jax.nn.sigmoid(a_sig * (coh - c0_eff))
            hybrid = hybrid + w * pred
            w_sum = w_sum + w

        hybrid = hybrid / (w_sum + 1e-6)

        # Wiener Blend
        Pxx = jnp.sum(wb * jnp.abs(delta_ref_k)**2)
        Pyy = jnp.sum(wb * jnp.abs(hybrid)**2)
        Pxy = jnp.sum(wb * jnp.real(delta_ref_k * jnp.conj(hybrid)))

        denom = (Pxx + Pyy - 2.0*Pxy) + 1e-12
        wx = (Pyy - Pxy) / denom
        wy = (Pxx - Pxy) / denom

        coh_band = Pxy / (jnp.sqrt(Pxx*Pyy + 1e-12) + 1e-12)
        gate_band = jax.nn.sigmoid(8.0 * (coh_band - 0.20))

        # Apply gate
        wx = gate_band * wx + (1.0 - gate_band) * 1.0
        wy = gate_band * wy * jax.nn.sigmoid(4.0 * (coh_band - 0.20))

        band_est = wx * delta_ref_k + wy * hybrid

        return (num_acc + wb * band_est, den_acc + wb, conf_acc + wb * jnp.clip(coh_band, 0.0,
1.0)), None

    init_fus = (jnp.zeros_like(delta_ref_k), jnp.zeros_like(k_mag), jnp.zeros_like(k_mag))
    (num_fus, den_fus, conf_fus), _ = jax.lax.scan(_fusion_scan, init_fus, band_centers)
```

```
        est_k = num_fus / (den_fus + 1e-12)
        conf_k = conf_fus / (den_fus + 1e-12)

        eta = jax.nn.sigmoid(hk_strength)
        damp = jax.nn.softplus(hk_strength)

        trust = w_hk * conf_k**2
        delta_fused_k = delta_ref_k + (eta * trust) * (est_k - delta_ref_k)
        delta_fused_k = delta_fused_k * jnp.exp(-damp * w_hk * (1.0 - conf_k)**2)
        delta_fused_k = jnp.where(k_sq == 0.0, 0.0, delta_fused_k)

        # --- Hard Low-k Lock ---
        final_k = jnp.where(k_mag <= 0.2, delta_base_k, delta_fused_k)
        final_k = jnp.where(k_sq == 0.0, 0.0, final_k)

        return (jnp.real(jnp.fft.ifftn(final_k)) + 1.0).astype(data.dtype)

    def _get_interpolated_displacement(pos, dx, dy, dz, nmesh, box):
        cell = box/nmesh
        x, y, z = pos[:,0]/cell, pos[:,1]/cell, pos[:,2]/cell
        ix, iy, iz = jnp.floor(x).astype(int)%nmesh, jnp.floor(y).astype(int)%nmesh,
    jnp.floor(z).astype(int)%nmesh
        fx, fy, fz = x - jnp.floor(x), y - jnp.floor(y), z - jnp.floor(z)
        ox, oy, oz = jnp.zeros_like(fx), jnp.zeros_like(fy), jnp.zeros_like(fz)
        for di in range(2):
            wx = jnp.where(di, fx, 1-fx)
            for dj in range(2):
                wy = jnp.where(dj, fy, 1-fy)
                for dk in range(2):
                    wz = jnp.where(dk, fz, 1-fz)
                    w = wx*wy*wz
                    ix0, iy0, iz0 = (ix+di)%nmesh, (iy+dj)%nmesh, (iz+dk)%nmesh
                    ox += dx[ix0, iy0, iz0] * w
                    oy += dy[ix0, iy0, iz0] * w
                    oz += dz[ix0, iy0, iz0] * w
        return ox, oy, oz

    def _differentiable_shift(pos, dx, dy, dz, nmesh, box):
        sx, sy, sz = _get_interpolated_displacement(pos, dx, dy, dz, nmesh, box)
        return jnp.mod(pos + jnp.stack([sx, sy, sz], axis=-1), box).astype(jnp.float32)

    def _differentiable_paint(pos, vals, nmesh, box):
        grid = jnp.zeros((nmesh, nmesh, nmesh), dtype=jnp.float32)
        cell = box/nmesh
        x, y, z = pos[:,0]/cell, pos[:,1]/cell, pos[:,2]/cell
        ix0, iy0, iz0 = jnp.floor(x).astype(int), jnp.floor(y).astype(int), jnp.floor(z).astype(int)
        dx, dy, dz = x-ix0, y-iy0, z-iz0
        for di in range(2):
            wx = jnp.where(di, dx, 1-dx)
            for dj in range(2):
                wy = jnp.where(dj, dy, 1-dy)
                for dk in range(2):
                    wz = jnp.where(dk, dz, 1-dz)
                    w = wx*wy*wz
                    grid = grid.at[(ix0+di)%nmesh, (iy0+dj)%nmesh, (iz0+dk)%nmesh].add(vals*w)
        return grid

    def _apply_differentiable_displacement(dens, dx, dy, dz, box, bias):
```

```
    n = dens.shape[0]
    H = box/n
    coords = jnp.stack(jnp.meshgrid(jnp.arange(n)*H, jnp.arange(n)*H, jnp.arange(n)*H,
indexing='ij'),
                       axis=-1).reshape(-1, 3).astype(jnp.float32)
    pos2 = _differentiable_shift(coords, dx, dy, dz, n, box)
    overd = dens.reshape(-1) - 1.0
    grid_d = _differentiable_paint(pos2, overd, n, box)
    grid_u = _differentiable_paint(pos2, jnp.ones_like(overd), n, box) - 1.0
    return (grid_d - grid_u)/bias + 1.0

# EVOLVE-BLOCK-END
```

This report was automatically generated using LLM-assisted analysis.