Motion Models in HGD: Physics, Implementation, and Recommendations

Analysis of Heterarchical Granular Dynamics Motion Models

November 11, 2025

Abstract

This document provides a comprehensive analysis of the motion model implementations in the Heterarchical Granular Dynamics (HGD) package. We characterize the physics, implementation strategies, computational features, and the presence or absence of inertia in each model. Based on this analysis, we provide recommendations for the `stream_core` function in `core.cpp`.

Contents

1 Introduction

The HGD package models granular flow by tracking the motion of voids through a granular medium. The fundamental approach is stochastic, where voids move according to probabilistic rules based on local conditions. Multiple implementations exist in the `motion/` subfolder, each with different trade-offs in terms of physical accuracy, computational efficiency, and implementation complexity.

1.1 Overview of HGD Physics

The core physics is based on:

- **Void migration**: Voids (NaN values in the grain size array) swap positions with solid particles
- **Probabilistic motion**: Swaps occur with probabilities based on gravity, local solid fraction, and particle sizes
- **Three directions**: Voids can move upward, left, or right
- **Mass conservation**: The total number of particles and voids is conserved

1.2 Key Parameters

- ν: Solid fraction (1 - void fraction)
- ν_{cs}: Critical solid fraction (typically 0.64, close-packed limit)
- s: Particle size (NaN represents voids)
- $\bar{s}$: Harmonic mean of particle sizes
- $\bar{s}^{-1}$: Inverse harmonic mean (for upward motion)
- α: Lateral diffusion coefficient
- P_u, P_l, P_r: Probabilities for upward, left, and right motion

2 Motion Model Implementations

2.1 d2q4_slow.py - Original Reference Implementation

2.1.1 Description

The original, straightforward Python implementation that processes each void individually. Named "d2q4" after the lattice Boltzmann nomenclature (2D with 4 directions, though only 3 are typically used).

2.1.2 Physics Implementation

Upward probability:

$$P_u = P_{u,\text{ref}} \frac{\bar{s}^{-1}_{i,j+1}}{s_{i,j+1,k}} \tag{1}$$

Lateral probabilities:

$$P_l = P_{lr,\text{ref}} \frac{s_{l,j,k}}{\bar{s}_{l,j}}, \quad P_r = P_{lr,\text{ref}} \frac{s_{r,j,k}}{\bar{s}_{r,j}} \tag{2}$$

where:

$$P_{u,\text{ref}} = v_y \frac{\Delta t}{\Delta y}, \quad P_{lr,\text{ref}} = \alpha P_{u,\text{ref}} \tag{3}$$

with $v_y = \sqrt{g\bar{s}}$ for average_size model or $v_y = \sqrt{2g\Delta y}$ for freefall model.
Slope stability: Lateral motion is prevented if the slope is stable:

$$\nu_{\text{neighbor}} - \nu_{\text{current}} \leq \text{scale\_ang} \cdot \nu_{cs} \tag{4}$$

2.1.3 Features

- **Inertia**: No
- **Parallelization**: No
- **Velocity tracking**: Simple counters (u, v) incremented when swaps occur
- **Conflict resolution**: Random selection among destination conflicts
- **Advection models**: Supports "average_size", "freefall", and "stress" (not implemented)

2.1.4 Pros

- Easy to understand and verify
- Clear physics implementation
- Good reference for debugging other implementations
- Flexible for experimentation

2.1.5 Cons

- Very slow ($O(n_x \cdot n_y \cdot n_m)$ with nested loops)
- No vectorization
- No inertia modeling
- Random index selection can be inefficient

2.2 d2q4_array.py - Vectorized Array Implementation

2.2.1 Description

A vectorized implementation using NumPy array operations to process all voids simultaneously for each direction of motion. Represents a significant performance improvement over the slow version.

2.2.2 Physics Implementation

The physics is similar to `d2q4_slow.py`, but implemented with array operations:
For each direction (up, left, right):

1. Calculate probabilities for all cells simultaneously
2. Use `np.roll` to access neighbor values
3. Apply stability criteria using boolean masks

4. Generate random numbers for all positions at once
5. Perform swaps where $P_{\text{random}} < P_{\text{swap}}$
6. Prevent overfilling by limiting swaps based on ν_{cs}

2.2.3 Features

- **Inertia**: No
- **Parallelization**: Implicit through NumPy (BLAS/LAPACK)
- **Velocity tracking**: Counters updated based on swap directions
- **Conflict resolution**: Random permutation before processing
- **Overfill prevention**: Checks against ν_{cs} and slope stability
- **Stress model support**: Can use stress-based velocities

2.2.4 Pros

- Much faster than `d2q4_slow.py` (10-100x depending on problem size)
- Vectorized operations utilize CPU efficiently
- Clear separation of physics per direction
- Easier to verify correctness by direction

2.2.5 Cons

- No inertia implementation
- Memory overhead from creating temporary arrays
- Sequential processing of directions (not truly parallel)
- Still relatively slow for large problems

2.3 d2q4_array_v2.py - Advanced Array Implementation with Inertia

2.3.1 Description

The most advanced Python implementation, extending `d2q4_array.py` with inertia support, multiple diffusion lengths, and advanced slope stability models. This represents the state-of-the-art physics implementation in Python.

2.3.2 Physics Implementation

Key enhancements:

1. **Inertia support:**

$$P_u = \frac{\Delta t}{\Delta y} U_{\text{dest}} \frac{\bar{s}^{-1}_{i,j+1}}{s_{i,j+1,k}} + \frac{\Delta t}{\Delta y} v_{i,j+1,k} \tag{5}$$

for upward motion, and:

$$P_{l/r} = \alpha U_{\text{dest}} s_{\text{dest}} \frac{\Delta t}{\Delta y^2} W + \frac{\Delta t}{\Delta y} u_{\text{dest}} \tag{6}$$

for lateral motion, where W is a diffusion weighting factor and u, v are the current velocities.

2. Multiple diffusion lengths: Allows voids to swap across multiple cells (up to `max_diff_swap_length`), with weighting:

$$W = \frac{n_{\max}(n_{\max}+1)(2n_{\max}+1)}{6} \tag{7}$$

3. Granular temperature damping:

$$\beta = \exp\left(-P_{\text{stab}}\frac{\Delta t}{\Delta x/u}\right) \tag{8}$$

4. Slope stability models:

- "gradient": Based on local solid fraction gradient
- "stress": Based on stress tensor from previous swaps

2.3.3 Features

- **Inertia**: Yes (optional, controlled by `p.inertia`)
- **Parallelization**: NumPy vectorization
- **Velocity tracking**: Full velocity field with momentum transfer
- **Conflict resolution**: Sophisticated `prevent_conflicts` and `prevent_overfilling` functions
- **Slope stability**: Multiple models (gradient/stress)
- **Advection models**: average_size, freefall, stress

2.3.4 Pros

- Most complete physics (inertia, stress, extended diffusion)
- Flexible and extensible
- Well-tested for research applications
- Handles complex phenomena (momentum transfer, granular temperature)
- Good for validation of other implementations

2.3.5 Cons

- Most complex implementation
- Highest memory usage
- Slower than C++ implementations
- Conflict resolution can be expensive for dense systems
- Some features (like stress model) may have issues (noted in comments)

2.4 d2q4_SA_array.py - Alternative Vectorized Implementation

2.4.1 Description

An alternative vectorized approach with a different formulation for probabilities and conflict resolution. This implementation explores a different mathematical formulation of the same physics.

2.4.2 Physics Implementation

Probability formulation: Pre-computes probability arrays for all three directions:

$$P_{ups} = P_{u,\text{ref}} \frac{\bar{s}^{-1}}{s} \text{ for upward} \tag{9}$$

$$P_{ls} = P_{lr,\text{ref}} \frac{s_{\text{left}}}{\bar{s}_{\text{left}}} \text{ for left} \tag{10}$$

$$P_{rs} = P_{lr,\text{ref}} \frac{s_{\text{right}}}{\bar{s}_{\text{right}}} \text{ for right} \tag{11}$$

Total probability:

$$P_{tot} = P_{ups} + P_{ls} + P_{rs} \tag{12}$$

Swap direction chosen based on random value compared to cumulative probabilities.
Conflict resolution: Uses explicit set operations to find and resolve conflicts:

- Find $A \cap B \cap C$ (three-way conflicts)
- Find pairwise conflicts $A \cap B$, $B \cap C$, $A \cap C$
- Randomly select which swap to keep

2.4.3 Features

- **Inertia**: No
- **Parallelization**: NumPy vectorization
- **Velocity tracking**: Not explicitly tracked
- **Conflict resolution**: Explicit set-based approach
- **Slope stability**: Angle-based criterion with `scale_ang` factor

2.4.4 Pros

- Mathematically elegant formulation
- Explicit conflict resolution is conceptually clear
- Pre-computation of probabilities
- Good for understanding alternative formulations

2.4.5 Cons

- No inertia
- Set operations for conflict resolution are slow
- Less tested than other implementations
- Velocity tracking is minimal
- Memory-intensive probability arrays

2.5 core.cpp - Optimized C++ Implementation

2.5.1 Description

The production C++ implementation, optimized for performance. This provides the best computational efficiency while maintaining accurate physics. It includes two main functions: `move_voids_core` (for void motion) and `stream_core` (for mass streaming based on velocities).

2.5.2 Physics Implementation

move_voids_core: Implements the core void motion algorithm similar to `d2q4_slow.py` but with:

- Precomputed neighbor indices
- Random shuffling of processing order
- Efficient storage using flattened arrays

stream_core: Implements mass advection based on mean velocities:

$$N_{u/l/r} = \lfloor \nu \cdot P_{u/l/r} \rfloor \tag{13}$$

where N is the number of particles to move in each direction. The function:

1. Computes movement probabilities from mean velocities
2. Determines number of particles to swap in each direction
3. Performs deterministic swaps (not probabilistic)
4. Prevents swaps into solid regions ($\nu \geq \nu_{cs}$)

2.5.3 Features

- **Inertia**: Parameter available but not fully implemented in stream_core
- **Parallelization**: Single-threaded (could be parallelized)
- **Velocity tracking**: Separate mean velocity arrays
- **Memory efficiency**: Uses std::vector, flattened indexing
- **Precomputation**: Neighbor indices cached in NeighborIndices struct
- **Random number generation**: Modern C++ std::mt19937

2.5.4 Pros

- Fast (10-100x faster than Python versions)
- Memory efficient
- Well-structured with helper functions
- Compiled optimization (-O3, -march=native)
- Low-level control over performance
- Pybind11 integration for easy Python access

2.5.5 Cons

- `stream_core` is incomplete (inertia not implemented)
- No vectorization (SIMD) used
- Single-threaded (no OpenMP)
- Less flexible than Python for experimentation
- Debugging is harder than Python

2.6 d2q4.cpp.dep - Deprecated C++ Implementation

2.6.1 Description

An earlier C++ implementation that has been deprecated. It used Eigen for linear algebra and had OpenMP hooks (commented out). This file provides historical context but should not be used for new work.

2.6.2 Features

- **Inertia**: Yes (parameter `p.inertia`)
- **Parallelization**: OpenMP hooks (commented out)
- **Eigen integration**: Used ArrayXXd types
- **Status**: Deprecated

2.6.3 Why Deprecated?

- Replaced by more efficient `core.cpp`
- Eigen dependency added complexity
- OpenMP parallelization was problematic (race conditions)
- Less maintainable than current implementation

Implementation	Relative Speed	Memory	Vectorization	Parallelization
d2q4_slow.py	1× (baseline)	Low	No	No
d2q4_array.py	10-50×	Medium	NumPy	Implicit
d2q4_array_v2.py	10-30×	High	NumPy	Implicit
d2q4_SA_array.py	5-20×	High	NumPy	Implicit
core.cpp	50-200×	Low	No	No
d2q4.cpp.dep	40-150×	Medium	Eigen	(Disabled)

Table 1: Performance comparison of motion model implementations

Implementation	Inertia	Stress	Multi-length	Slope Model	Status
d2q4_slow.py	No	Partial	No	Gradient	Reference
d2q4_array.py	No	Yes	No	Gradient	Active
d2q4_array_v2.py	Yes	Yes	Yes	Both	Active
d2q4_SA_array.py	No	No	No	Angle	Experimental
core.cpp	Partial	No	No	Gradient	Production
d2q4.cpp.dep	Yes	No	No	Gradient	Deprecated

Table 2: Physics features in each implementation

3 Comparative Analysis

3.1 Performance Comparison

3.2 Physics Feature Comparison

4 Inertia Implementation Details

4.1 What is Inertia in HGD?

In the HGD context, "inertia" refers to momentum carried by particles/voids as they move. Without inertia:

- Each void swap is independent
- Velocities are just counters (statistics)
- No momentum conservation
- Instant response to force changes

With inertia:

- Particles carry velocity u, v
- Velocity swaps with particle during void migration
- Probabilities depend on current velocity
- Gradual acceleration/deceleration
- More realistic dynamics

4.2 Implementations with Inertia

d2q4_array_v2.py:

- Most complete inertia implementation
- Velocities stored in 3D arrays $u[i, j, k]$, $v[i, j, k]$
- Velocities swap with particles: $(u[i, j, k], u[\text{dest}]) \leftarrow (u[\text{dest}], u[i, j, k])$
- Contribution to swap probability: $P+ = (\Delta t/\Delta y) \cdot v_{\text{dest}}$
- Velocity of new void set to zero
- Solid regions have zero velocity enforced

d2q4.cpp.dep (deprecated):

- Similar approach to d2q4_array_v2.py
- Used for inertial flows
- Velocities tracked and swapped with particles
- Commented out collision detection

core.cpp:

- Parameter `p.inertia` exists but not used in `stream_core`
- `move_voids_core` updates velocities but doesn't use them for probability
- `stream_core` uses mean velocities but as deterministic advection, not probabilistic
- Incomplete inertia implementation

5 Recommendations for stream_core

5.1 Current Issues

The current `stream_core` function in `core.cpp` has several limitations:

1. **No probabilistic motion**: Uses deterministic particle counts $N = \lfloor \nu \cdot P \rfloor$
2. **Limited inertia**: Doesn't fully incorporate momentum into swap probabilities
3. **Simplified physics**: Lacks features from advanced Python implementations
4. **Sequential processing**: No attempt at vectorization or parallelization

5.2 Recommended Approach

Based on the analysis, I recommend the following for improving `stream_core`:

5.2.1 Option 1: Minimal Upgrade (Recommended for Production)

Goal: Add inertia support while maintaining performance and simplicity.

Changes:

1. Add velocity fields $u[i, j, k]$ and $v[i, j, k]$ to the function signature
2. Include inertia contribution in swap probabilities:

$$P_u = \text{mask}(i, j+1) \cdot (v_{\text{mean}}[i, j+1] + v[i, j+1, k]) \frac{\Delta y}{\Delta t} \tag{14}$$

3. Swap velocities with particles during streaming
4. Make the swap probabilistic (not deterministic) when inertia is enabled

Physics basis: Follow `d2q4_array_v2.py` equations but in deterministic streaming context.

Pros:

- Maintains C++ performance
- Adds critical inertia feature
- Relatively simple to implement and test
- Compatible with existing code structure

Cons:

- Still missing advanced features (stress, multi-length)
- Not as complete as d2q4_array_v2.py

5.2.2 Option 2: Full Feature Parity with d2q4_array_v2.py

Goal: Implement all features from the most advanced Python implementation.

Changes:

1. Full inertia with momentum conservation
2. Multiple diffusion lengths
3. Stress-based slope stability
4. Granular temperature tracking
5. Advanced conflict resolution

Pros:

- Most accurate physics
- Enables cutting-edge research
- Future-proof

Cons:

- Complex implementation
- Higher memory usage
- May sacrifice some performance
- Longer development time
- Harder to maintain

5.2.3 Option 3: Hybrid Approach (Recommended for Long-term)

Goal: Create a flexible C++ implementation that can switch between modes.

Strategy:

1. Keep simple streaming for non-inertial flows
2. Add inertia mode with momentum transfer
3. Use template metaprogramming or compile-time flags
4. Optimize each mode separately

Example:

```
template<bool WithInertia>
void stream_core_impl(...) {
    if constexpr (WithInertia) {
        // Full inertia implementation
    } else {
        // Simple streaming
    }
}
```

Pros:

- Best performance for each mode
- Flexible for different applications
- No runtime overhead for non-inertial cases

Cons:

- More complex code structure
- Need to maintain two code paths

5.3 Specific Implementation Details for Option 1

For the recommended minimal upgrade, here's the specific approach:

5.3.1 Modified Function Signature

```
void stream_core(
    const std::vector<double>& u_mean,
    const std::vector<double>& v_mean,
    View3<double> u,  // Add full velocity field
    View3<double> v,  // Add full velocity field
    View3<double> s,
    const View2<const uint8_t>& mask,
    std::vector<double>& nu,
    const Params& p);
```

5.3.2 Key Algorithm Changes

1. Probability computation with inertia:

```
double P_u = mask(i, j + 1) ?
    (v_mean[idx_up] + (p.inertia ? v(i, j+1, k) : 0.0))
    * dy_over_dt : 0;
```

2. Velocity swapping:

```
if (p.inertia && found) {
    // Swap velocities with particle
    double tmp_u = u(i, j, k);
    u(i, j, k) = u(dest[0], dest[1], k);
    u(dest[0], dest[1], k) = tmp_u;

    double tmp_v = v(i, j, k);
    v(i, j, k) = v(dest[0], dest[1], k);
    v(dest[0], dest[1], k) = tmp_v;

    // Set void velocity to zero
    u(i, j, k) = 0.0;
    v(i, j, k) = 0.0;
}
```

3. Probabilistic vs deterministic:

- Non-inertial: Keep deterministic $N = \lfloor \nu \cdot P \rfloor$
- Inertial: Use probabilistic swap like `move_voids_core`

5.4 Testing and Validation

To validate any changes to `stream_core`:

1. **Unit tests**: Test individual components (probability calculation, velocity swap)
2. **Comparison tests**: Compare with `d2q4_array_v2.py` for simple cases
3. **Conservation tests**: Verify mass, momentum conservation
4. **Physical tests**: Run standard benchmarks (collapse, segregation, hourglass)
5. **Performance tests**: Ensure no significant performance regression

6 Conclusion

The HGD motion models span a range of complexity and performance:

- **For understanding physics**: Use `d2q4_slow.py`
- **For Python research**: Use `d2q4_array_v2.py`
- **For production simulations**: Use `core.cpp`

The main gap in the current implementation is proper inertia support in `stream_core`. The recommended approach is Option 1 (minimal upgrade), which adds essential inertia features while maintaining the performance and simplicity of the C++ implementation.

Key recommendations for `stream_core`:

1. Add full velocity field support
2. Include inertia contribution to swap probabilities
3. Swap velocities with particles when inertia is enabled
4. Consider making swaps probabilistic in inertia mode
5. Validate against `d2q4_array_v2.py`
6. Maintain performance for non-inertial cases

Future work could include stress-based models, multi-length diffusion, and parallelization, following Option 3 (hybrid approach) for maximum flexibility.

A Mathematical Notation Summary

Symbol	Meaning
ν	Solid fraction
ν_{cs}	Critical solid fraction (close packing)
s	Particle size (NaN for voids)
$\bar{s}$	Harmonic mean of particle sizes
$\bar{s}^{-1}$	Inverse harmonic mean
P_u	Upward swap probability
P_l	Leftward swap probability
P_r	Rightward swap probability
α	Lateral diffusion coefficient
v_y	Characteristic velocity scale
g	Gravitational acceleration
Δt	Time step
$\Delta x, \Delta y$	Grid spacing
u, v	Horizontal and vertical velocities
n_x, n_y	Grid dimensions
n_m	Number of particles per cell (microstructural coordinate)

Table 3: Mathematical notation used throughout this document

B Code Structure Reference

B.1 File Organization

```
HGD/motion/
|-- __init__.py
|-- bindings_py.cpp       # Pybind11 bindings
|-- core.cpp              # Production C++ implementation
|-- core.h                # Header for core.cpp
|-- d2q4_slow.py          # Reference Python implementation
|-- d2q4_array.py         # Vectorized Python
|-- d2q4_array_v2.py      # Advanced Python with inertia
|-- d2q4_SA_array.py      # Alternative formulation
|-- d2q4.cpp.dep          # Deprecated C++ implementation
```

```
|-- helpers.cpp           # Helper functions (deprecated)
+-- helpers.h             # Helper headers (deprecated)
```

B.2 Key Functions

core.cpp:

- `move_voids_core`: Main void motion with probabilistic swaps
- `stream_core`: Mass streaming based on velocities
- `compute_solid_fraction_core`: Calculate ν
- `compute_mean_core`: Calculate $\bar{s}$
- `compute_s_inv_bar_core`: Calculate $\bar{s}^{-1}$

Python implementations:

- `move_voids`: Main entry point for void motion
- Various operator functions in `HGD.operators`
- Stress calculation in `HGD.stress` (for d2q4_array_v2.py)