Vicsek Model Simulation on Sphere

Alireza Same

August 10, 2025

Abstract

This project introduces a high-performance simulation of the Vicsek model on a spherical manifold, designed to investigate the statistical physics of collective motion in a topologically closed environment. By constraining self-propelled particles to a spherical surface, the simulation eliminates boundary effects inherent in Euclidean models, offering a more realistic framework for studying long-range order and emergent phenomena. The implementation utilizes 3D Cartesian vectors for robust physics calculations, including geodesic motion for particle updates and parallel transport for accurate velocity comparisons, thereby avoiding singularities and ensuring geometrically correct dynamics. The framework supports configurable noise models, enabling systematic studies of the order-disorder phase transition, cluster formation, and non-equilibrium dynamics. This ensemble-based approach provides a powerful tool for large-scale investigations into the complex collective behaviors of active matter on curved surfaces, contributing to a deeper understanding of emergent phenomena in biological and physical systems

1 Spherical Manifold

In this model, each "bird" is represented by its position and orientation on a sphere of radius R. The position of a particle is defined by a point on the spherical surface, while its velocity vector is always tangent to the sphere at that specific point, maintaining a constant speed s. Consequently, the state of each particle can be fully described by a compact set of three parameters: (θ, ϕ, α). Here, (θ, ϕ) denote the standard spherical coordinates defining the particle's location on the sphere's surface. The third parameter, α, specifies the direction of the velocity vector within the local tangent plane at the particle's position. This angle α is measured relative to the local $\hat{e}_\theta$ (tangential) basis vector. For a clearer visualization, see Figure 1.
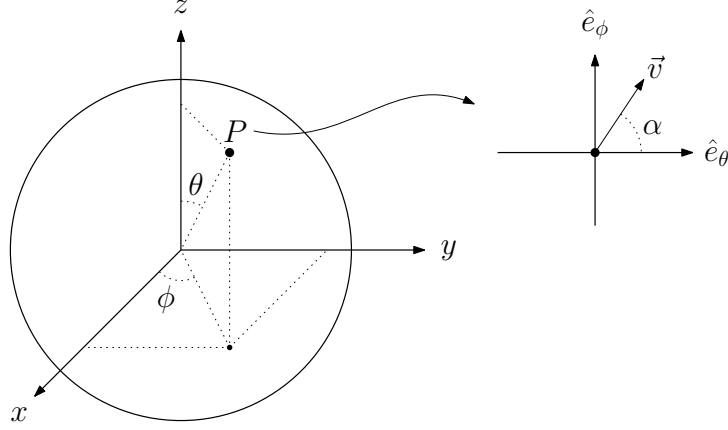
Figure 1: Representation of a particle on the spherical manifold.

To convert the spherical coordinates θ, ϕ, α to 3D Cartesian coordinates, we use the following transformations: The Cartesian coordinates (x, y, z) of a particle located at (θ, ϕ) on the sphere of radius R are given by:

$$
\begin{cases}
x = R \sin\theta \cos\phi \\
y = R \sin\theta \sin\phi \\
z = R \cos\theta
\end{cases}
,
\quad
\begin{cases}
\hat{e}_r = \sin\theta \cos\phi\, \hat{i} + \sin\theta \sin\phi\, \hat{j} + \cos\theta\, \hat{k} \\
\hat{e}_\theta = \cos\theta \cos\phi\, \hat{i} + \cos\theta \sin\phi\, \hat{j} - \sin\theta\, \hat{k} \\
\hat{e}_\phi = -\sin\phi\, \hat{i} + \cos\phi\, \hat{j}
\end{cases}
\tag{1}
$$

The velocity vector $\vec{v}$ can be expressed as follows:

$$
\vec{v} = s \left(\cos\alpha\, \hat{e}_\theta + \sin\alpha\, \hat{e}_\phi \right)
\tag{2}
$$

1.1 Great-circle Distance

In this simulation, determining the distance between two birds is crucial. We employ the great-circle distance, representing the shortest path along the Earth's surface. While typically associated with spherical coordinates, calculating this distance is simplified using Cartesian coordinates. The angle between two points on the sphere can be derived from the dot product of their position vectors. The great-circle distance d_{GC} between two points on the sphere is then given by:

$$
d_{GC}\left(P_1, P_2\right) = R \cdot \arccos\left(\frac{\vec{r_1} \cdot \vec{r_2}}{R^2}\right)
\tag{3}
$$

In our simulation, we acknowledge two singular scenarios for inter-bird distance. The first, a zero-distance case $(d_{GC} = 0)$, signifies a collision. The second, the antipodal case $(d_{GC} = \pi R)$, represents birds on diametrically opposite sides of the sphere, precluding visual contact. Both singularities will be excluded from the simulation. Therefore, the distance between any two birds will always satisfy $0 < d_{GC} < \pi R$. It is important to note that a more appropriate maximal distance for certain applications, such as

visibility calculations, might be $d_{GC} = \frac{\pi}{2}R$, which significantly alters the scope of potential interactions.

1.2 Parallel Transport

To determine the average direction of neighboring birds, it is necessary to parallel transport their respective velocities to the base bird's position before averaging. This procedure ensures the meaningfulness of the resulting average, see Figure 2.
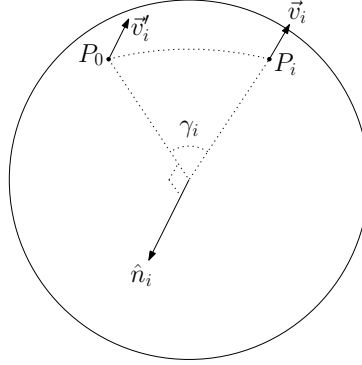

Figure 2: Parallel transport of velocities

Let P_0 represent the base bird, and $P_1, P_2, \ldots, P_n$ denote the n birds within its visual range. The parallel transport of the velocity vector $\vec{v}_i$ of bird P_i to the base bird P_0 is accomplished via a rotation about the appropriate axis. This rotation will be calculated using Rodrigues' rotation formula, as given by Equation (5).

$$\hat{n}_i = \frac{\vec{r}_i \times \vec{r}_0}{\|\vec{r}_i \times \vec{r}_0\|}, \quad \gamma_i = \arccos\left(\frac{\vec{r}_i \cdot \vec{r}_0}{R^2}\right) \tag{4}$$

$$\vec{v}_i' = \vec{v}_i \cos\gamma_i + (\hat{n}_i \times \vec{v}_i)\sin\gamma_i + \hat{n}_i(\hat{n}_i \cdot \vec{v}_i)(1 - \cos\gamma_i) \tag{5}$$

As evident, the parallel transport operation encounters a singularity when dealing with antipodal points. Consequently, such cases are excluded from consideration. Following the parallel transport of all neighboring birds' velocities, these vectors are averaged to yield a new resultant velocity vector direction. It is important to note that this resultant vector may, in certain instances, be a zero vector. In such an eventuality, the current velocity vector of the base bird will be retained without modification. Subsequently, the derived velocity vector is normalized to maintain a constant speed s. We will denote this by $\vec{v}_{\text{ave}}$. Finally, the simulation's inherent noise component is incorporated into this normalized velocity vector.

3

1.3 Noise Addition (Rotation)

Upon determining the new direction, a small random rotation is applied. This rotation is performed within the tangent space at the current point, thereby preserving the tangency of the velocity vector. The magnitude of this rotational angle is sampled from a pre-defined distribution, which will be elaborated upon subsequently. Utilizing Rodrigues' rotation formula once more for this operation, we obtain:

$$\vec{v}_{\text{transport}} = \vec{v}_{\text{ave}} \cos \alpha_n + (\hat{r}_0 \times \vec{v}_{\text{ave}}) \sin \alpha_n + \hat{r}_0 (\hat{r}_0 \cdot \vec{v}_{\text{ave}})(1 - \cos \alpha_n)$$

1.4 Time step

With the new velocity vector established, the bird's position and velocity are updated in two stages. First, the bird's position is advanced along a geodesic according to the new velocity vector, using a time step dt. Subsequently, this new velocity vector is parallel transported to the updated position, similar to previous operations. This yields the bird's new position and velocity vector, which serve as the initial conditions for the subsequent simulation step. This inherent Markovian property, where the current state depends solely on the previous state, offers significant computational advantages.

We know that velocity is perpendicular to position vector (tangency) this will simplify the geodesic equation to:

$$\vec{r}(t + dt) = \cos \left(\frac{\|\vec{v}_{\text{transport}}\| dt}{R} \right) \vec{r}(t) + \frac{R}{s} \sin \left(\frac{\|\vec{v}_{\text{transport}}\| dt}{R} \right) \vec{v}_{\text{transport}} \tag{6}$$

$$\vec{v}(t + dt) = \cos \left(\frac{\|\vec{v}_{\text{transport}}\| dt}{R} \right) \vec{v}_{\text{transport}} - \frac{s}{R} \sin \left(\frac{\|\vec{v}_{\text{transport}}\| dt}{R} \right) \vec{r}(t) \tag{7}$$

Two considerations are paramount during time-stepping. Firstly, to mitigate floating-point errors, the angular displacement, $s \cdot dt$, must remain sufficiently small; specifically, it should not exceed $\frac{\pi}{2} R$, as larger values would constitute an excessive jump. Secondly, it is imperative to manually verify and normalize the magnitudes of both the position and velocity vectors. This normalization procedure will effectively mitigate the accumulation of floating-point errors.

2 Random Variables

The simulation necessitates the generation of random values for both ensemble initialization and noise injection. This section will detail the methodologies employed for random number generation, including the specific probability distributions utilized.

2.1 Ensemble generation

To establish the initial conditions for our simulation, we must generate an ensemble of birds with randomly distributed positions. Utilizing spherical coordinates simplifies this process. For truly random sets, we employ uniform distributions for angle generation. The azimuthal angle, ϕ, and the angle for the initial velocity direction, α, are directly sampled uniformly from 0 to 2π. However, generating the polar angle, θ, requires a more nuanced approach. Simply sampling θ uniformly from 0 to π would result in an disproportionate concentration of points near the poles. To achieve a truly uniform distribution of points across the spherical surface, we instead generate $\cos\theta$ uniformly from -1 to 1, and then derive θ from this value. This method ensures an even spatial distribution of birds, avoiding artificial clustering at the poles.

$$\begin{cases} \theta = \arccos(U[-1,1]) \\ \phi \sim U[0, 2\pi] \\ \alpha \sim U[0, 2\pi] \end{cases}$$

For ensemble generation, we sample these three angles. A critical step involves verifying that no pre-existing bird is in close proximity to the newly generated position, thereby preventing unwanted initial collisions. Once N birds have been successfully created, this set constitutes a single ensemble, which is then saved. This process is repeated to generate a total of M ensembles, facilitating subsequent ensemble averaging.

2.2 Noise Generation

To introduce noise, we apply a rotational angle, α_n, ranging from $-\pi$ to π, where a value of zero signifies no rotation. This angle is sampled from a normal distribution centered at zero with a variance of η^2, where η represents the noise strength parameter, ranging from zero to infinity. As η increases (approaching 4π), the distribution of α_n effectively becomes uniform, leading to complete randomness. Conversely, a value of $\eta = 0$ introduces no randomness. There exists a critical value of η that corresponds to a phase transition within the system.

$$\alpha_n \sim \mathcal{N}(0, \eta^2), \quad 0 \le \eta < 4\pi$$

3 Parameter Sensitivity Analysis

The primary objective of this simulation is to identify phase changes within the system. Given our ensemble-based approach, a parameter sensitivity analysis is essential. For each simulation run, the following parameters will be examined individually:

- **Radius** (R): This parameter holds no true influence on the simulation's intrinsic

behavior. Its selection is primarily to mitigate numerical issues. Beyond this, the radius lacks significance, as it does not even incorporate scaling. Its sole effect is to impart a dimension to the neighbor distance parameter, rendering it largely irrelevant to the core dynamics.

- **Speed** (s): Similar to the radius, the speed parameter is also largely irrelevant. Its primary function is to establish the system's time scale in conjunction with the radius. As previously discussed, the time step must be sufficiently small to maintain logical consistency; thus, speed is a free parameter that imposes a bound on dt. Beyond this, it holds no further significance in the simulation.

- **Neighbor Distance** (d): This parameter holds significant importance, though it functions more as a classifying variable than a continuous one for trend analysis. For instance, a value of $\pi R/2$ would correspond to a mean-field approximation of the model. Consequently, we only require approximately four distinct values of this parameter to discern its impact on the system's behavior.

- **Noise Strength** (η): This stands as the most crucial parameter in our simulation, as we aim to observe its continuous impact on phase transitions. We contend that noise strength is the primary catalyst for these transitions within the system. By systematically varying η, we can observe the system's progression from ordered to disordered states, thereby revealing the critical noise level at which these phase changes manifest. Our simulation specifically targets the identification of the numerous first-order phase transitions expected in this system.

- **Number of Birds** (N): To investigate finite-size scaling effects, we must vary the number of birds in the system. Similar to the parameter d, changes in this parameter are expected to reveal only minor trends. Given that this system does not exhibit continuous phase transitions, we do not anticipate observing significant finite-size scaling behavior around any noise parameter, as critical exponents are absent.

- **Number of Ensembles** (M): This parameter is crucial for ensuring statistical significance in our results. By increasing the number of ensembles, we can obtain a more reliable estimate of the average behavior of the system. However, it is important to note that beyond a certain point, increasing M yields diminishing returns in terms of improved accuracy.

- **Time Step** (dt): The time step is a critical parameter that must be chosen carefully to ensure the stability and accuracy of the simulation. A small dt is necessary to avoid numerical instabilities, especially when dealing with high-speed particles or large-radius spheres. However, excessively small time steps can lead to increased

computational costs without significant benefits. Therefore, it is essential to find an optimal balance for *dt* that maintains the integrity of the simulation while minimizing computational overhead.

4   Implementation Details

The simulation is implemented in the Rust programming language. The library's functionality is organized into the following submodules:

- Core Physics:

    - vector: Provides optimized 3D vector mathematics crucial for spherical geometry.

    - bird: Defines the fundamental particle representation and their associated physics.

- Simulation Engine:

    - ensemble: Facilitates the generation of initial conditions, particularly for uniform particle distribution on the sphere.

    - simulation: Houses the high-performance, parallelized simulation loop and state management.

    - analysis: Implements metrics for collective behavior, such as order parameters and cluster analysis.

- Utilities and Analysis:

    - io: Manages data persistence and serialization.

    - cli: Defines the command-line interface for direct execution.

4.1   Vector Module

The `Vec3` structure is designed to represent a vector in a 3D Cartesian coordinate system.

Listing 1: `Vec3` struct

```
#[derive(Clone, Copy, serde::Serialize, serde::Deserialize)]
pub struct Vec3 {
    pub x: f64,
    pub y: f64,
    pub z: f64,
}
```

The `Vec3` structure provides a rich set of methods for vector manipulation, creation, and mathematical operations. These methods are categorized below for clarity.

4.1.1 Constructors and Static Methods

- **`new(x: f64, y: f64, z: f64)`**

  Creates a new 3D vector with the specified x, y, and z components. This is the primary way to instantiate a `Vec3`.

- **`zero()`**

  Returns a zero vector $(0, 0, 0)$. Useful as an origin point or additive identity.

- **`x_hat()`**

  Returns the unit vector along the positive X-axis $(1, 0, 0)$.

- **`y_hat()`**

  Returns the unit vector along the positive Y-axis $(0, 1, 0)$.

- **`z_hat()`**

  Returns the unit vector along the positive Z-axis $(0, 0, 1)$.

4.1.2 Magnitude and Normalization Methods

- **`norm_squared() : f64`**

  Calculates the squared magnitude $(x^2 + y^2 + z^2)$ of the vector. More efficient than `norm()` as it avoids a square root.

- **`norm() : f64`**

  Calculates the magnitude (Euclidean length) of the vector $(\sqrt{x^2 + y^2 + z^2})$.

- **`normalize() : Self`**

  Returns a unit vector (magnitude $= 1$) in the same direction. Returns the zero vector if the original vector has zero or near-zero magnitude.

4.1.3 Vector Operations

- **`dot(&self, other: &Self) : f64`**

  Calculates the dot product (scalar product) with another vector. Measures how much two vectors point in the same direction.

- **`cross(&self, other: &Self) : Self`**

  Calculates the cross product with another vector. Produces a vector perpendicular to both input vectors, following the right-hand rule.

- **`angle_between(&self, other: &Self) : f64`**

  Calculates the angle in radians between this vector and another. The result is in the range $[0, \pi]$.

- **`rotate_around(&self, axis: &Self, angle: f64) : Option<Self>`**

  Rotates this vector around a specified *normalized* axis by a given angle in radians. Returns `Some(Vec3)` on success, `None` if the axis is invalid (e.g., zero vector or not normalized).

4.1.4 Utility Functions

- **`approx_eq(&self, other: &Self, epsilon: f64) : bool`**

  Checks if this vector is approximately equal to another within a given `epsilon` tolerance for each component, due to floating-point precision.

- **`Add<Vec3> for Vec3`** (+) Overloads the addition operator (+) for vector-vector addition.

- **`Sub<Vec3> for Vec3`** (-) Overloads the subtraction operator (-) for vector-vector subtraction.

- **`Mul<f64> for Vec3`** (*) Overloads the multiplication operator (*) for scalar multiplication (`Vec3 * f64`).

- **`Mul<Vec3> for f64`** (*) Overloads the multiplication operator (*) for commutative scalar multiplication (`f64 * Vec3`).

- **`Div<f64> for Vec3`** (/) Overloads the division operator (/) for scalar division (`Vec3 / f64`).

- **`Neg for Vec3`** (-) Overloads the unary negation operator (-) to reverse the vector's direction.

- **`PartialEq for Vec3`** (==) Implements equality comparison (==) for `Vec3` instances. Note that due to floating-point arithmetic, `approx_eq` is generally preferred for meaningful comparisons.

4.2 Bird Module

The `Bird` struct represents a particle in the simulation.

Listing 2: `Bird` struct

```rust
#[derive(Clone, Copy, serde::Serialize, serde::Deserialize)]
pub struct Bird {
    pub position: Vec3,
    pub velocity: Vec3,
}
```

Given that the struct implements the `Copy` and `Clone` traits, it can be efficiently passed by value or duplicated, thereby avoiding ownership complexities. Spherical operations are optimized and performed through its inherent methods.

4.2.1 Constructors and Static Methods

- **new(position: Vec3, velocity: Vec3) (private)**

  Creates a new bird with the specified position and velocity vectors. It is the responsibility of the caller to ensure that the position is on the sphere's surface and the velocity is tangent to the sphere at that point. *Not safe to use*

- **from_spherical(radius: f64, theta:f64, phi: f64, speed: f64, alpha: f64)**

  Returns a new bird instance with its position and velocity initialized based on spherical coordinates. This is the recommended way to create a bird in the simulation.

- **random_angle_noise(order_parameter: f64) : f64 (private)**

  This function produces normally distributed random angles used to introduce noise into bird velocity directions, simulating environmental perturbations and individual behavioral variations in flocking systems.

4.2.2 Physics Methods

- **distance_from(&self, other: &Self, radius: f64) : f64**

  Calculates the geodesic distance between two birds on a sphere surface.

- **parallel_transport_velocity(&self, base: &Self) : Vec3**

  Performs parallel transport of this bird velocity to another bird position. This is essential for averaging velocities in the simulation.

- **`add_noise(averaged: Vec3, base: &Self, order_parameter: f64) : Vec3`**

  Adds angular noise to an averaged velocity vector around a reference position. This method applies random rotational noise to a velocity vector, typically used after averaging neighbor velocities in flocking algorithms. The noise is applied as a rotation around the local normal (position) vector. it uses `random_angle_noise` to generate the noise angle, which is sampled from a normal distribution centered at zero with a variance of η^2.

- **`move_on_sphere(&self, dt: f64, radius: f64, speed: f64) : Self`**

  Moves this bird along the sphere surface using geodesic motion and parallel transport. This method implements complete bird movement on a sphere surface, updating both the bird position via geodesic motion and its velocity via parallel transport. The movement preserves the constraint that the bird remains on the sphere surface and maintains the velocity tangent nature to the sphere.

4.3   Ensemble Module

This module facilitates the generation of an initial ensemble of "birds," uniformly distributed on a spherical surface for flocking simulations. It employs a rejection sampling methodology to ensure a minimum geodesic separation between particles, thereby preventing initial overlaps while maintaining spatial homogeneity. Particle positions are determined using spherical coordinates, with polar angles derived from a uniform distribution of their cosine to guarantee a truly uniform surface coverage. The generated ensemble is then transmitted via a Multi-Producer, Single-Consumer (MPSC) channel, enabling asynchronous processing and decoupling the generation logic from subsequent data handling. For instance, to generate 1000 birds on a unit sphere with a speed of 2.0 and a minimum separation of 0.1, one would invoke `ensemble::generate(1000, 1.0, 2.0, 0.1, tx).unwrap();`, where `tx` is the sender half of an MPSC channel.

5   Results

Hopefully someday