TwinkleSwarm

Illuminated Drone Swarm Coordination

Stochastic Batman

January 11, 2026

Abstract

This document provides a mathematical foundation for `TwinkleSwarm`, a simulation framework for coordinating illuminated drone swarms. We present the differential equations governing drone motion, analyze their behavior, and describe the algorithms for extracting target formations from images and motion patterns from videos. The framework combines control theory, particle dynamics, and computer vision to achieve collision-free formation control. `Claude Sonnet 4.5` was used to correct grammar, format math quickly, and check for errors.

1 Introduction

`TwinkleSwarm` simulates coordinated drone swarms that form visual patterns extracted from images or follow motion extracted from videos. Each drone is modeled as a particle with position and velocity, subject to control forces that drive it toward target positions while avoiding collisions with neighboring drones.

The system addresses three sequential challenges:

1. **Static Formation**: Moving drones from arbitrary initial positions to form a handwritten pattern
2. **Shape Transition**: Smoothly transitioning the swarm from one formation to another
3. **Dynamic Tracking**: Following moving objects in video while preserving the swarm's shape

2 Mathematical Model

2.1 State Representation

Each drone i (where $i = 1, 2, \ldots, N$) is characterized by:

- $\mathbf{x}_i(t) \in \mathbb{R}^3$: Position at time t
- $\mathbf{v}_i(t) \in \mathbb{R}^3$: Velocity at time t

The complete swarm state at time t is the collection of all drone states: $(\mathbf{x}_1, \mathbf{v}_1, \mathbf{x}_2, \mathbf{v}_2, \ldots, \mathbf{x}_N, \mathbf{v}_N)$.

2.2 Governing Equations

The motion of each drone follows a second-order system with velocity saturation and inter-drone repulsion. We present the Initial Value Problem (IVP) formulation for time-forward simulation.

2.2.1 Initial Value Problem (IVP) Formulation

The IVP formulation describes how drones evolve from known initial conditions:

$$\dot{\mathbf{x}}_i(t) = \mathbf{v}_i(t) \cdot \min\left(1, \frac{v_{\max}}{\|\mathbf{v}_i(t)\|}\right) \tag{1}$$

$$\dot{\mathbf{v}}_i(t) = \frac{1}{m}\left[k_p(\mathbf{T}_i(t) - \mathbf{x}_i(t)) + \sum_{j \neq i} \mathbf{f}_{\text{rep}}(\mathbf{x}_i(t), \mathbf{x}_j(t)) - k_d \mathbf{v}_i(t)\right] \tag{2}$$

with initial conditions:

$$\mathbf{x}_i(0) = \mathbf{x}_{i,0}, \quad \mathbf{v}_i(0) = \mathbf{v}_{i,0} \tag{3}$$

Intuition: Equation (1) updates position based on velocity, but caps the speed at $v_{\max}$ (a physical constraint). Equation (2) describes how velocity changes over time, driven by three forces:

- **Attraction to target**: $k_p(\mathbf{T}_i - \mathbf{x}_i)$ pulls the drone toward its assigned target $\mathbf{T}_i$
- **Repulsion from neighbors**: $\sum_{j \neq i} \mathbf{f}_{\text{rep}}$ pushes drones apart to avoid collisions
- **Damping**: $-k_d \mathbf{v}_i$ reduces velocity over time, preventing oscillation and ensuring smooth convergence

Parameters:

- $m > 0$: Drone mass (constant)
- $k_p > 0$: Proportional gain controlling attraction strength
- $k_d > 0$: Damping coefficient controlling convergence speed
- $v_{\max} > 0$: Maximum allowable velocity
- $\mathbf{T}_i(t) \in \mathbb{R}^3$: Time-varying target position for drone i

2.2.2 Repulsive Force Model

To prevent collisions, drones repel each other when too close. The repulsive force between drones i and j is:

$$\mathbf{f}_{\text{rep}}(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} k_{\text{rep}} \cdot \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|^3} & \text{if } \|\mathbf{x}_i - \mathbf{x}_j\| < R_{\text{safe}} \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{4}$$

where $k_{\text{rep}} > 0$ is the repulsion gain and $R_{\text{safe}} > 0$ is the safety radius.

Intuition: The force follows an inverse-cube law, meaning it grows rapidly as drones get closer. The direction $\frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|}$ is the unit vector pointing from drone j to drone i, so the force pushes drone i away from drone j. The cubic dependence ensures strong short-range repulsion while allowing free movement when drones are far apart.

2.3 Velocity Fields and Saturation

2.3.1 What is a Velocity Field?

A velocity field is a function $\mathbf{V} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$ that assigns a velocity vector to each point in space and time:

$$\mathbf{V}(\mathbf{x}, t) = \begin{bmatrix} V_x(\mathbf{x}, t) \\ V_y(\mathbf{x}, t) \\ V_z(\mathbf{x}, t) \end{bmatrix} \tag{5}$$

Intuition: Think of a velocity field like a wind map. At every location $\mathbf{x}$ and time t, the field tells you "if you were at this position, here's how fast and in what direction you should be moving". For drone swarms following video motion, the velocity field describes how the target shape is moving through space.

Unlike static targets $\mathbf{T}_i$ which tell drones "where to go", a velocity field tells them "how to move". For dynamic tracking: when following a moving object, we don't just want to know where the object is now, but how it's moving so we can move with it.

2.3.2 The Saturation Problem

In practice, velocity fields extracted from videos can contain arbitrarily large velocities (computer vision algorithms are not perfect, yet). A fast-moving object might have $\|\mathbf{V}(\mathbf{x}, t)\| \gg v_{\max}$, where $v_{\max}$ is the physical velocity limit of our drones.

Why We Care About Saturation:

- **Physical constraints**: Real drones have maximum speeds determined by motor capabilities, battery power, and aerodynamics.
- **Safety**: Exceeding velocity limits can lead to loss of control or collisions.
- **Simulation validity**: If we allow unbounded velocities in simulation, our results won't transfer to real hardware.
- **Numerical stability**: Very large velocities can cause integration schemes to become unstable or require impractically small time steps.

The Naive Approach Fails: Simply using $\dot{\mathbf{v}}_i = \mathbf{V}(\mathbf{x}_i, t) - \mathbf{v}_i$ doesn't respect velocity limits. Even if we clip drone velocities after integration, the control law itself doesn't "know" about the constraint, leading to poor tracking and potential instability.

2.3.3 Saturated Velocity Field

To address this, we define the saturated velocity field:

$$\mathbf{V}_{\text{sat}}(\mathbf{x}, t) = \begin{cases} \mathbf{V}(\mathbf{x}, t) \cdot \min\left(1, \frac{v_{\max}}{\|\mathbf{V}(\mathbf{x}, t)\|}\right) & \text{if } \|\mathbf{V}(\mathbf{x}, t)\| > 0 \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{6}$$

How It Works:

- If $\|\mathbf{V}(\mathbf{x}, t)\| \leq v_{\max}$, then $\min(1, \frac{v_{\max}}{\|\mathbf{V}\|}) = 1$ and $\mathbf{V}_{\text{sat}} = \mathbf{V}$ (no change)
- If $\|\mathbf{V}(\mathbf{x}, t)\| > v_{\max}$, then $\min(1, \frac{v_{\max}}{\|\mathbf{V}\|}) = \frac{v_{\max}}{\|\mathbf{V}\|} < 1$, which scales the vector down

The result is that $\mathbf{V}_{\text{sat}}$ always satisfies $\|\mathbf{V}_{\text{sat}}(\mathbf{x}, t)\| \leq v_{\max}$.

Geometric Interpretation: Saturation preserves direction but limits magnitude. If you visualize the velocity field as a collection of arrows, saturation "clips" the long arrows to maximum length while leaving short arrows unchanged. The field retains its qualitative structure (which direction motion is flowing) while respecting physical constraints.

Example: Suppose $v_{\max} = 5$ m/s and at some point $\mathbf{V} = (12, 9, 0)^T$ m/s. Then $\|\mathbf{V}\| = 15$ m/s. The saturated version is:

$$\mathbf{V}_{\text{sat}} = (12, 9, 0)^T \cdot \frac{5}{15} = (4, 3, 0)^T \text{ m/s}$$

The direction (north-east) is preserved, but the speed is reduced from 15 to 5 m/s.

2.4 Velocity Field Tracking (for Dynamic Scenarios)

When tracking motion from video, we use a velocity field formulation instead of fixed targets:

$$\dot{\mathbf{x}}_i(t) = \mathbf{v}_i(t) \cdot \min\left(1, \frac{v_{\max}}{\|\mathbf{v}_i(t)\|}\right) \tag{7}$$

$$\dot{\mathbf{v}}_i(t) = \frac{1}{m}\left[k_v(\mathbf{V}_{\text{sat}}(\mathbf{x}_i(t), t) - \mathbf{v}_i(t)) + \sum_{j \neq i} \mathbf{f}_{\text{rep}}(\mathbf{x}_i(t), \mathbf{x}_j(t)) - k_d \mathbf{v}_i(t)\right] \tag{8}$$

where $\mathbf{V}(\mathbf{x}, t)$ is a velocity field extracted from video (see Section 3), and $\mathbf{V}_{\text{sat}}$ is its saturated version.

Intuition: Instead of moving toward fixed target positions, drones now try to match their velocities to a field $\mathbf{V}(\mathbf{x}, t)$ that describes how the target shape is moving. The term $k_v(\mathbf{V}_{\text{sat}} - \mathbf{v}_i)$ drives each drone's velocity toward the field velocity at its current position. This allows the swarm to track dynamic motion while maintaining its formation.

2.5 Analytical Solutions

2.5.1 Single Drone, No Repulsion

Consider a single drone ($N = 1$) with no repulsive forces, moving toward a fixed target $\mathbf{T}$ (independent of time):

$$\dot{\mathbf{x}}(t) = \mathbf{v}(t) \tag{9}$$

$$\dot{\mathbf{v}}(t) = \frac{1}{m}[k_p(\mathbf{T} - \mathbf{x}(t)) - k_d \mathbf{v}(t)] \tag{10}$$

This is a linear system. Define the error $\mathbf{e}(t) = \mathbf{x}(t) - \mathbf{T}$. Then:

$$\dot{\mathbf{e}}(t) = \mathbf{v}(t) \tag{11}$$

$$\dot{\mathbf{v}}(t) = -\frac{k_p}{m}\mathbf{e}(t) - \frac{k_d}{m}\mathbf{v}(t) \tag{12}$$

To demonstrate that this system represents a damped harmonic oscillator, we can reduce the coupled first-order equations into a single second-order differential equation.

Differentiating the first equation with respect to time gives the acceleration:

$$\ddot{\mathbf{e}}(t) = \dot{\mathbf{v}}(t) \tag{13}$$

Substituting the expression for from the second equation yields:

$$\ddot{\mathbf{e}}(t) = -\frac{k_p}{m}\mathbf{e}(t) - \frac{k_d}{m}\mathbf{v}(t) = -\frac{k_p}{m}\mathbf{e}(t) - \frac{k_d}{m}\dot{\mathbf{e}}(t) \tag{14}$$

Rearranging results in the standard second-order linear homogeneous ODE:

$$\ddot{\mathbf{e}}(t) + \frac{k_d}{m}\dot{\mathbf{e}}(t) + \frac{k_p}{m}\mathbf{e}(t) = \mathbf{0} \tag{15}$$

This is a damped harmonic oscillator. Assume a solution of the form $e(t) = e^{\lambda t}$, then as $e^{\lambda t} \neq 0$, divide by $e^{\lambda t}$. Then, the characteristic equation is:

$$\lambda^2 + \frac{k_d}{m}\lambda + \frac{k_p}{m} = 0 \tag{16}$$

with roots:

$$\lambda_{1,2} = \frac{-k_d \pm \sqrt{k_d^2 - 4mk_p}}{2m} \tag{17}$$

Case 1: **Overdamped** ($k_d^2 > 4mk_p$). Both roots are real and negative, leading to exponential decay without oscillation.

Case 2: **Critically Damped** ($k_d^2 = 4mk_p$). A repeated negative root, yielding the fastest convergence without overshoot.

Case 3: **Underdamped** ($k_d^2 < 4mk_p$). Complex conjugate roots with negative real part, resulting in damped oscillations around the target.

Stability: Because k_d and m are positive, the real part of λ is always negative, which ensures that $\mathbf{e}(t) \to \mathbf{0}$ as $t \to \infty$ and the drones actually reach their targets.

The general solution for the underdamped case is:

$$\mathbf{e}(t) = e^{-\alpha t}(\mathbf{A}\cos(\omega t) + \mathbf{B}\sin(\omega t)) \tag{18}$$

where $\alpha = \frac{k_d}{2m}$ (decay rate) and $\omega = \frac{\sqrt{4mk_p - k_d^2}}{2m}$ (oscillation frequency). The coefficients $\mathbf{A}$ and $\mathbf{B}$ depend on initial conditions.

2.5.2 Energy Analysis

Define the total energy of the system as:

$$E(t) = \sum_{i=1}^{N}\left[\frac{1}{2}m\|\mathbf{v}_i(t)\|^2 + \frac{1}{2}k_p\|\mathbf{x}_i(t) - \mathbf{T}_i(t)\|^2\right] \tag{19}$$

The first term is kinetic energy, the second is potential energy from the attraction to targets. Taking the time derivative (ignoring repulsion and time-varying targets for simplicity):

$$\frac{dE}{dt} = \sum_{i=1}^{N} [m\mathbf{v}_i \cdot \dot{\mathbf{v}}_i + k_p(\mathbf{x}_i - \mathbf{T}_i) \cdot \dot{\mathbf{x}}_i] \tag{20}$$

Substituting the dynamics:

$$\frac{dE}{dt} = \sum_{i=1}^{N} [\mathbf{v}_i \cdot (k_p(\mathbf{T}_i - \mathbf{x}_i) - k_d\mathbf{v}_i) + k_p(\mathbf{x}_i - \mathbf{T}_i) \cdot \mathbf{v}_i] \tag{21}$$

$$= \sum_{i=1}^{N} \left[-k_d\|\mathbf{v}_i\|^2\right] \tag{22}$$

$$= -k_d \sum_{i=1}^{N} \|\mathbf{v}_i\|^2 \leq 0 \tag{23}$$

Interpretation: Energy decreases over time due to damping, ensuring the system converges to a steady state where all drones are stationary at their targets.

3 Video Processing and Motion Extraction

This section describes how we extract motion information from video to drive the drone swarm. The key idea is to compute a *velocity field* that describes how pixels move between consecutive frames, then use this field to guide drone motion.

3.1 The Motion Extraction Problem

Given a video sequence $\{I_1, I_2, \ldots, I_T\}$ where each $I_t : \Omega \to [0, 1]$ is a grayscale image defined on domain $\Omega \subset \mathbb{R}^2$, we want to find a velocity field $\mathbf{V} : \Omega \times \mathbb{R} \to \mathbb{R}^2$ that describes how the image content moves over time.

Intuition: Imagine watching a ball roll across the screen. At each moment, different pixels correspond to the ball's surface. As the ball moves, these pixels appear to "flow" across the image. The velocity field captures this flow: at each pixel location and time, it tells us which direction and how fast the visual content is moving.

3.2 Optical Flow

Optical flow is the pattern of apparent motion of objects in a visual scene caused by relative motion between the observer and the scene. We compute optical flow between consecutive frames to extract this motion.

3.2.1 The Brightness Constancy Assumption

The fundamental assumption underlying most optical flow methods is **brightness constancy**: a point in the scene maintains constant brightness as it moves between frames. Mathematically, if a pixel at location (x, y) in frame t moves to location $(x + u, y + v)$ in frame $t + 1$, then:

$$I_t(x, y) = I_{t+1}(x + u, y + v) \tag{24}$$

where (u, v) is the displacement vector (the flow).

Why This Makes Sense: If you're tracking a red dot moving across a white background, the dot doesn't change color as it moves - only its position changes. The brightness constancy assumption formalizes this: the intensity value "travels" with the motion.

When It Fails: This assumption breaks down when:

- Lighting changes (shadows appear/disappear)
- Objects rotate (different surfaces become visible)
- Occlusions occur (objects pass behind each other)
- Motion blur is significant

Despite these limitations, brightness constancy works well for small displacements and consistent lighting.

3.2.2 Deriving the Optical Flow Equation

To convert the brightness constancy assumption into a usable equation, we perform a first-order Taylor expansion of $I_{t+1}(x+u, y+v)$ around (x, y):

$$I_{t+1}(x+u, y+v) \approx I_{t+1}(x, y) + \frac{\partial I_{t+1}}{\partial x}u + \frac{\partial I_{t+1}}{\partial y}v \tag{25}$$

This is valid when displacements (u, v) are small. Now, assuming the image changes smoothly over time, we can approximate:

$$I_{t+1}(x, y) \approx I_t(x, y) + \frac{\partial I_t}{\partial t} \cdot \Delta t \tag{26}$$

where Δt is the time between frames (typically $\Delta t = 1$ frame). For small time intervals, $\frac{\partial I_{t+1}}{\partial x} \approx \frac{\partial I_t}{\partial x}$ and similarly for y. Substituting the Taylor expansion into the brightness constancy equation:

$$I_t(x, y) = I_t(x, y) + \frac{\partial I_t}{\partial t} + \frac{\partial I_t}{\partial x}u + \frac{\partial I_t}{\partial y}v \tag{27}$$

Canceling $I_t(x, y)$ from both sides:

$$\frac{\partial I_t}{\partial x}u + \frac{\partial I_t}{\partial y}v + \frac{\partial I_t}{\partial t} = 0 \tag{28}$$

This is the **optical flow constraint equation**. In compact notation:

$$\nabla I_t \cdot \mathbf{v}_f + I'_t = 0 \tag{29}$$

where:

- $\nabla I_t = \left(\frac{\partial I_t}{\partial x}, \frac{\partial I_t}{\partial y}\right)^T$ is the spatial gradient (direction of brightness change)
- $\mathbf{v}_f = (u, v)^T$ is the velocity (flow) vector
- $I'_t = \frac{\partial I_t}{\partial t}$ is the temporal derivative (brightness change over time)

Geometric Interpretation: Equation (28) says that if brightness is constant along motion, then the rate of brightness change over time (I'_t) must be balanced by motion along the brightness gradient ($\nabla I_t \cdot \mathbf{v}_f$). If a pixel moves toward brighter regions at the right speed, the temporal darkening cancels the spatial brightening.

The Aperture Problem: This is one equation with two unknowns (u and v). Geometrically, we can only determine the component of flow perpendicular to image edges - motion parallel to edges is invisible. This is called the *aperture problem*. To solve it, we need additional constraints.

3.3 Lucas-Kanade Method

The Lucas-Kanade method resolves the aperture problem by assuming the flow is **constant in a local neighborhood** $\mathcal{N}$ around each pixel. This is reasonable for small patches where all pixels belong to the same moving object.

3.3.1 Formulation

For a neighborhood $\mathcal{N}$ centered at pixel (x_0, y_0), we assume (u, v) is constant across all pixels $(x, y) \in \mathcal{N}$. The optical flow constraint should hold at every pixel, but due to noise and modeling errors, we solve it in a least-squares sense:

$$\min_{u,v} \sum_{(x,y)\in\mathcal{N}} \left[\frac{\partial I_t}{\partial x}(x,y) \cdot u + \frac{\partial I_t}{\partial y}(x,y) \cdot v + \frac{\partial I_t}{\partial t}(x,y)\right]^2 \tag{30}$$

Intuition: We're fitting a single velocity (u, v) that best explains the brightness changes across all pixels in the patch. Some pixels might not fit perfectly (due to noise), but the least-squares solution minimizes overall error.

3.3.2 Matrix Form

Let $I_x = \frac{\partial I_t}{\partial x}$, $I_y = \frac{\partial I_t}{\partial y}$, and $I_t = \frac{\partial I_t}{\partial t}$. The least-squares problem can be written in matrix form. Define:

$$A = \begin{bmatrix} I_x(x_1, y_1) & I_y(x_1, y_1) \\ I_x(x_2, y_2) & I_y(x_2, y_2) \\ \vdots & \vdots \\ I_x(x_n, y_n) & I_y(x_n, y_n) \end{bmatrix}, \quad \mathbf{b} = -\begin{bmatrix} I_t(x_1, y_1) \\ I_t(x_2, y_2) \\ \vdots \\ I_t(x_n, y_n) \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix} \tag{31}$$

where (x_i, y_i) are the n pixels in $\mathcal{N}$. The least-squares solution minimizes $\|A\mathbf{v} - \mathbf{b}\|^2$, which is:

$$\mathbf{v} = (A^T A)^{-1} A^T \mathbf{b} \tag{32}$$

Computing $A^T A$ and $A^T \mathbf{b}$:

$$A^T A = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}, \quad A^T \mathbf{b} = -\begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{33}$$

where all sums are over $(x, y) \in \mathcal{N}$. Therefore:

$$\begin{bmatrix} u \\ v \end{bmatrix} = -\begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{34}$$

3.3.3 Interpreting the Structure Tensor

The matrix $A^T A$ is called the **structure tensor** or **second moment matrix**. Its properties determine whether the flow can be reliably computed:

- **Large eigenvalues in both directions**: The patch contains texture in multiple orientations (e.g., a corner). Flow is well-determined.
- **One large eigenvalue**: The patch contains an edge. Flow perpendicular to the edge is determined, but parallel flow is ambiguous (aperture problem).
- **Small eigenvalues**: The patch is uniform (constant intensity). No reliable flow can be computed.

In practice, we only compute flow at pixels where $A^T A$ is well-conditioned, typically by checking that its smallest eigenvalue exceeds a threshold.

3.4 Farneback Method

The Farneback method is a dense optical flow algorithm that approximates the neighborhood of each pixel with polynomial expansions. Unlike Lucas-Kanade which assumes constant flow in a local neighborhood, Farneback models the image intensity as a second-order polynomial and estimates displacement by comparing polynomial coefficients between frames.

3.4.1 Polynomial Expansion Model

The fundamental idea is to represent the image intensity $I(\mathbf{x})$ in a neighborhood around each point using a quadratic polynomial:

$$I(\mathbf{x}) \approx \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c \tag{35}$$

where $\mathbf{A}$ is a symmetric 2×2 matrix, $\mathbf{b}$ is a 2×1 vector, and c is a scalar constant. This expansion is computed using a weighted least-squares fit over the neighborhood, typically with a Gaussian weighting function.

3.4.2 Displacement Estimation

Consider two consecutive frames I_1 and I_2. Under the assumption of brightness constancy and small displacement $\mathbf{d} = (u, v)^T$, we have:

$$I_2(\mathbf{x}) = I_1(\mathbf{x} - \mathbf{d}) \tag{36}$$

If we expand I_1 around $\mathbf{x}$ using the polynomial model:

$$I_1(\mathbf{x} - \mathbf{d}) \approx (\mathbf{x} - \mathbf{d})^T \mathbf{A}_1 (\mathbf{x} - \mathbf{d}) + \mathbf{b}_1^T (\mathbf{x} - \mathbf{d}) + c_1 \tag{37}$$

$$= \mathbf{x}^T \mathbf{A}_1 \mathbf{x} + (\mathbf{b}_1 - 2\mathbf{A}_1 \mathbf{d})^T \mathbf{x} + (c_1 - \mathbf{b}_1^T \mathbf{d} + \mathbf{d}^T \mathbf{A}_1 \mathbf{d}) \tag{38}$$

Comparing this with the polynomial expansion of I_2 at point $\mathbf{x}$:

$$I_2(\mathbf{x}) = \mathbf{x}^T \mathbf{A}_2 \mathbf{x} + \mathbf{b}_2^T \mathbf{x} + c_2 \tag{39}$$

Equating coefficients gives the system:

$$\mathbf{A}_2 = \mathbf{A}_1 \tag{40}$$

$$\mathbf{b}_2 = \mathbf{b}_1 - 2\mathbf{A}_1\mathbf{d} \tag{41}$$

$$c_2 = c_1 - \mathbf{b}_1^T\mathbf{d} + \mathbf{d}^T\mathbf{A}_1\mathbf{d} \tag{42}$$

From the second equation, we can solve for the displacement:

$$\mathbf{d} = -\frac{1}{2}\mathbf{A}_1^{-1}(\mathbf{b}_2 - \mathbf{b}_1) \tag{43}$$

provided $\mathbf{A}_1$ is invertible (i.e., the neighborhood has sufficient texture).

3.4.3 Practical Implementation

In practice, the Farneback algorithm as implemented in OpenCV's `cv2.calcOpticalFlowFarneback()`.

Advantages over Lucas-Kanade:

- Produces dense flow fields (flow at every pixel)
- More robust to larger displacements
- Better handling of textureless regions through global constraints

Disadvantages:

- Computationally more expensive than sparse methods
- Requires careful parameter tuning
- May oversmooth small motions

For `TwinkleSwarm`, we use the Farneback method as implemented in OpenCV because it provides dense flow fields that are essential for tracking entire shapes in video frames.

3.5 Computing Spatial and Temporal Derivatives

To apply the Lucas-Kanade method, we need to compute I_x, I_y, and I_t numerically from discrete images.

3.5.1 Spatial Gradients

The spatial derivatives are approximated using finite differences. Common choices include:

Central difference (second-order accurate):

$$I_x(x, y) \approx \frac{I_t(x+1, y) - I_t(x-1, y)}{2} \tag{44}$$

Sobel operator (more robust to noise):

$$I_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I_t \tag{45}$$

where $*$ denotes convolution. The Sobel operator applies smoothing (averaging in the y-direction) while differentiating in x, reducing sensitivity to noise.

3.5.2 Temporal Derivative

The temporal derivative is computed as:

$$I_t(x, y) \approx I_{t+1}(x, y) - I_t(x, y) \tag{46}$$

For better accuracy, some implementations average over neighboring pixels:

$$I_t(x, y) \approx \frac{1}{9} \sum_{\Delta x \in \{-1,0,1\}} \sum_{\Delta y \in \{-1,0,1\}} [I_{t+1}(x + \Delta x, y + \Delta y) - I_t(x + \Delta x, y + \Delta y)] \tag{47}$$

3.6 Velocity Field Construction for Drone Control

The optical flow computation gives us a 2D velocity field $\mathbf{V}_{\text{raw}}(x, y, t) = (u(x, y, t), v(x, y, t))^T$ in pixel coordinates. To use this for controlling drones in physical 3D space, we perform several transformations:

Step 1: Filtering and Smoothing

Raw optical flow often contains noise and outliers. We apply spatial smoothing (e.g., Gaussian blur) to obtain a smoother field:

$$\mathbf{V}_{\text{smooth}} = G_\sigma * \mathbf{V}_{\text{raw}} \tag{48}$$

where G_σ is a Gaussian kernel with standard deviation σ.

Step 2: Coordinate Transformation

Image coordinates have origin at the top-left corner with y increasing downward, while our drone coordinate system has origin at the center with y increasing upward. We transform:

$$\begin{bmatrix} x_{\text{drone}} \\ y_{\text{drone}} \end{bmatrix} = \begin{bmatrix} x_{\text{pixel}} - \frac{w}{2} \\ \frac{h}{2} - y_{\text{pixel}} \end{bmatrix} \tag{49}$$

where $w \times h$ is the image resolution. The velocity field transforms accordingly:

$$\mathbf{V}_{\text{drone}}(x, y, t) = \begin{bmatrix} u(x, y, t) \\ -v(x, y, t) \end{bmatrix} \tag{50}$$

Step 3: Physical Scaling

Pixel velocities (pixels/frame) must be converted to physical velocities (meters/second). Let s be the scale factor and f the frame rate:

$$\mathbf{V}_{\text{physical}} = s \cdot f \cdot \mathbf{V}_{\text{drone}} \tag{51}$$

For example, if $s = 0.01$ m/pixel and $f = 30$ fps, a flow of $(10, 5)$ pixels/frame becomes $(3.0, 1.5)$ m/s.

Step 4: 3D Extension

Since videos provide only 2D motion, we extend to 3D by setting the z-component to zero:

$$\mathbf{V}(x, y, z, t) = \begin{bmatrix} V_x(x, y, t) \\ V_y(x, y, t) \\ 0 \end{bmatrix} \tag{52}$$

This keeps the swarm motion in the plane parallel to the image.

Step 5: Spatial Interpolation

Drones can be at arbitrary positions, not just pixel centers. We use bilinear interpolation to evaluate the velocity field at any point (x, y):

Let (x, y) fall in the grid cell with corners at (x_0, y_0), (x_1, y_0), (x_0, y_1), (x_1, y_1). Define normalized coordinates:

$$\alpha = \frac{x - x_0}{x_1 - x_0}, \quad \beta = \frac{y - y_0}{y_1 - y_0} \tag{53}$$

Then:

$$\begin{aligned}\mathbf{V}(x, y, t) =&(1 - \alpha)(1 - \beta)\mathbf{V}(x_0, y_0, t) + \alpha(1 - \beta)\mathbf{V}(x_1, y_0, t) \\ &+ (1 - \alpha)\beta\mathbf{V}(x_0, y_1, t) + \alpha\beta\mathbf{V}(x_1, y_1, t)\end{aligned} \tag{54}$$

Step 6: Saturation

Finally, we apply velocity saturation (as described in Section 2.3.3) to ensure $\|\mathbf{V}\| \leq v_{\max}$:

$$\mathbf{V}_{\text{sat}}(\mathbf{x}, t) = \begin{cases}\mathbf{V}(\mathbf{x}, t) \cdot \min\left(1, \frac{v_{\max}}{\|\mathbf{V}(\mathbf{x}, t)\|}\right) & \text{if } \|\mathbf{V}(\mathbf{x}, t)\| > 0 \\ \mathbf{0} & \text{otherwise}\end{cases} \tag{55}$$

3.7 Practical Considerations

3.7.1 Multi-Scale Optical Flow

For large displacements (when objects move many pixels between frames), the Taylor expansion underlying equation (28) breaks down. The solution is to compute flow at multiple scales:

1. Create an image pyramid: $\{I^{(0)}, I^{(1)}, \ldots, I^{(L)}\}$ where $I^{(l+1)}$ is a downsampled version of $I^{(l)}$
2. Compute flow at level L (where motion is small)
3. Upsample the flow and use it to warp $I^{(l-1)}$
4. Compute residual flow at level $l - 1$
5. Repeat until reaching the original resolution

This is called the **coarse-to-fine** or **pyramidal** approach.

3.7.2 Dense vs. Sparse Flow

- **Dense flow**: Computed at every pixel. Good for tracking entire shapes but computationally expensive.
- **Sparse flow**: Computed only at feature points (corners, edges). Faster but may miss motion in textureless regions.

For drone swarms, dense flow is preferable because we want to track the entire visible shape, not just isolated points.

3.7.3 Alternative Methods

While we focus on Lucas-Kanade, other optical flow methods include:

- **Horn-Schunck**: Adds a global smoothness constraint, encouraging similar flow in neighboring pixels
- **Deep learning methods** (FlowNet, PWC-Net): Neural networks trained on large datasets

OpenCV provides implementations of these methods through `cv2.calcOpticalFlowFarneback()` and related functions.

4 Assignment Algorithms and Spatial Data Structures

4.1 Hungarian (Munkres) Algorithm

The Hungarian algorithm, also known as the Munkres assignment algorithm, solves the assignment problem in polynomial time. Given N drones and N target positions, we need to assign each drone to a unique target to minimize the total squared distance traveled.

4.1.1 Problem Formulation

Let $C \in \mathbb{R}^{N \times N}$ be a cost matrix where C_{ij} represents the cost of assigning drone i to target j. In our case:

$$C_{ij} = \|\mathbf{x}_i(0) - \mathbf{T}_j\|^2 \tag{56}$$

where $\mathbf{x}_i(0)$ is the initial position of drone i and $\mathbf{T}_j$ is the position of target j.
We seek a permutation $\pi : \{1, \ldots, N\} \to \{1, \ldots, N\}$ that minimizes:

$$\sum_{i=1}^{N} C_{i,\pi(i)} \tag{57}$$

4.1.2 Algorithm Steps

The Hungarian algorithm operates on the cost matrix through a series of row and column operations:

1. **Row reduction**: For each row i, subtract the minimum element from all elements in that row:
$$C_{ij} \leftarrow C_{ij} - \min_k C_{ik} \quad \forall j \tag{58}$$
2. **Column reduction**: For each column j, subtract the minimum element from all elements in that column:
$$C_{ij} \leftarrow C_{ij} - \min_k C_{kj} \quad \forall i \tag{59}$$
3. **Cover zeros with minimum lines**: Find the minimum number of horizontal and vertical lines needed to cover all zeros in the matrix. If N lines are needed (where N is the matrix size), an optimal assignment exists among the zeros. Otherwise, proceed to step 4.
4. **Create additional zeros**: Let δ be the smallest uncovered element. Subtract δ from all uncovered elements and add δ to all elements covered twice. Return to step 3.

4.1.3 Complexity and Implementation

The Hungarian algorithm has time complexity $O(N^3)$, which is efficient for moderate swarm sizes (up to several hundred drones). We use the implementation from SciPy's `linear_sum_assignment` function, which is based on the Jonker-Volgenant algorithm, a more efficient variant of the Hungarian algorithm.

Intuition: The algorithm systematically reduces the cost matrix while maintaining the property that the optimal assignment remains unchanged. The row and column operations preserve the relative ordering of assignment costs, allowing us to find zero-cost assignments that correspond to optimal matchings.

4.2 K-d Trees for Spatial Queries

A k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in k-dimensional space. For drone swarms in 3D space ($k = 3$), k-d trees enable efficient neighbor searches essential for computing repulsive forces.

4.2.1 Data Structure

A k-d tree is a binary tree where each node represents a point in $\mathbb{R}^k$. Each non-leaf node splits the space along one of the coordinate axes using a hyperplane perpendicular to that axis. The tree is constructed recursively:

1. Choose the dimension with the greatest spread of points
2. Select the median point along that dimension
3. Create a node with this point
4. Recursively build left subtree with points having coordinate less than median
5. Recursively build right subtree with points having coordinate greater than median

4.2.2 Range Search Algorithm

To find all points within distance R_{safe} of a query point $\mathbf{q}$, we perform a recursive search. The algorithm can be described as follows:

Algorithm 1 K-d Tree Range Search

```
procedure RangeSearch(node, q, R_safe)
    if node is null then
        return
    end if
    d ← ‖node.point − q‖
    if d < R_safe then
        Add node.point to results
    end if
    split_dim ← node.split_dimension
    split_val ← node.point[split_dim]
    q_val ← q[split_dim]
    if q_val − R_safe < split_val then
        RangeSearch(node.left, q, R_safe)
    end if
    if q_val + R_safe > split_val then
        RangeSearch(node.right, q, R_safe)
    end if
end procedure
```

4.2.3 Complexity Analysis

- **Construction**: $O(N \log N)$ on average, $O(N^2)$ worst-case for poorly distributed points
- **Query**: $O(\log N)$ for nearest neighbor, $O(N^{1-1/k} + m)$ for range queries returning m points
- **Memory**: $O(N)$ to store the tree structure

4.2.4 Application to Drone Repulsion

For N drones, naive pairwise distance checking would require $O(N^2)$ operations. Using k-d trees with range search reduces this to approximately $O(N \log N)$ for well-distributed drones. In `TwinkleSwarm`, we use SciPy's `KDTree` implementation which provides efficient methods like `query_pairs` to find all pairs within a specified distance.

Hybrid Approach: For small swarms ($N < 50$), we use brute-force (but using `numba`'s JIT) pairwise checking which has lower constant overhead. For larger swarms, we switch to k-d tree based neighbor search. This hybrid approach optimizes performance across different swarm sizes.

5 Numerical Integration

5.1 Runge-Kutta Method (RK45)

For the numerical integration of the drone dynamics equations, we employ the Runge-Kutta method of order 5(4), commonly referred to as RK45 or the Dormand-Prince method. This method provides an adaptive step-size control mechanism that balances accuracy and computational efficiency.

5.1.1 General Runge-Kutta Formulation

A general s-stage Runge-Kutta method for solving the initial value problem $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$ is given by:

$$\mathbf{k}_i = \mathbf{f}\left(t_n + c_i \Delta t, \mathbf{y}_n + \Delta t \sum_{j=1}^{s} a_{ij} \mathbf{k}_j\right), \quad i = 1, \ldots, s \tag{60}$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \Delta t \sum_{i=1}^{s} b_i \mathbf{k}_i \tag{61}$$

where c_i, a_{ij}, and b_i are method-specific coefficients typically organized in a Butcher tableau.

5.1.2 RK45 (Dormand-Prince) Coefficients

The RK45 method uses 7 stages to produce both a 5th-order accurate solution $\mathbf{y}_{n+1}$ and a 4th-order accurate solution $\hat{\mathbf{y}}_{n+1}$, which are used for error estimation and step-size control. The Butcher tableau for RK45 is:

$$\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}$$

The bottom row with b_i coefficients gives the 5th-order solution, while the $\hat{b}_i$ coefficients (second bottom row) give the 4th-order solution used for error estimation.

5.1.3 Step-size Control

The local error estimate is computed as:

$$\mathbf{e}_{n+1} = \|\mathbf{y}_{n+1} - \hat{\mathbf{y}}_{n+1}\| \tag{62}$$

Given a desired tolerance τ, the optimal step size for the next step is determined by:

$$\Delta t_{\text{new}} = \Delta t_{\text{old}} \cdot \min\left(f_{\max}, \max\left(f_{\min}, \beta \cdot \left(\frac{\tau}{\mathbf{e}_{n+1}}\right)^{1/5}\right)\right) \tag{63}$$

where $\beta \approx 0.9$ is a safety factor, and $f_{\min}$, $f_{\max}$ limit the step size change (typically $f_{\min} = 0.1$, $f_{\max} = 5.0$).

5.1.4 Implementation in `TwinkleSwarm`

In our implementation, we use SciPy's `solve_ivp` function with the `RK45` method, specifying relative and absolute tolerances:

$$\texttt{rtol} = 5 \times 10^{-3}, \quad \texttt{atol} = 1 \times 10^{-4} \tag{64}$$

These tolerances provide a good balance between accuracy and computational efficiency for our drone dynamics. The adaptive step-size control ensures that we take larger steps when the system evolves smoothly and smaller steps during rapid transients (e.g., when drones are close to each other and repulsive forces are strong).

5.1.5 Advantages for Drone Simulation

The RK45 method is particularly well-suited for drone swarm simulation because:

- **Adaptive step size**: Automatically adjusts to the local dynamics, ensuring stability during close interactions while maintaining efficiency during steady flight
- **High accuracy**: 5th-order local accuracy provides precise trajectory computation
- **Error control**: Built-in error estimation allows for controlled trade-off between accuracy and speed
- **Widely used**: Well-tested implementation in SciPy with excellent numerical properties

6 Parameter Selection

Choosing appropriate parameters is crucial for desired behavior:

- **Mass** m: Can be normalized to $m = 1$ for simplicity
- **Proportional gain** k_p: Higher values increase attraction strength but may cause oscillation. Typical range: 1–10
- **Damping** k_d: Should satisfy $k_d \geq 2\sqrt{mk_p}$ for critical/overdamping. Typical range: 2–5
- **Repulsion gain** $k_{\mathbf{rep}}$: Should be strong enough to prevent collisions but not so strong as to disrupt formation. Typical range: 0.1–1
- **Safety radius** $R_{\mathbf{safe}}$: Should be at least twice the drone's physical radius. Typical value: 0.5–2 meters
- **Maximum velocity** $v_{\max}$: Hardware-dependent, typically 1–10 m/s

References

[1] `TwinkleSwarm` repository: `https://github.com/Stochastic-Batman/TwinkleSwarm`

[2] B.D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision", *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, 1981.

[3] G. Farnebäck, "Two-Frame Motion Estimation Based on Polynomial Expansion", *Proceedings of the 13th Scandinavian Conference on Image Analysis*, 2003.

[4] OpenCV Documentation: `https://docs.opencv.org/4.x/`

[5] J. Munkres, "Algorithms for the assignment and transportation problems", *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[6] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM*, vol. 18, issue 9, pp. 509–517, 1975.

[7] J.R. Dormand and P.J. Prince, "A family of embedded Runge-Kutta formulae", *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, 1980.