Common Components: Shared Mathematical Foundations, Numerical Methods, and Code Mapping

Illuminated Drone Show Simulation (NP 2025 Final Project)

January 30, 2026

1 Why this section is vital to the requirements

The project statement requires the student to (i) formulate each sub-problem as an IVP/BVP, (ii) clearly describe all inputs and parameters, (iii) specify numerical methods and justify them, (iv) provide test cases and limitations, and (v) explicitly mention AI usage. A dedicated *Common Components* section is vital because it demonstrates that Subtasks 1–3 are consistent extensions of a single dynamical system and numerical pipeline:

(i) **Physics engine:** spring–damper attraction to targets, short-range repulsion, and velocity saturation;

(ii) **Numerical solver:** RK4 time integration with fixed Δt and multiple substeps per rendered frame;

(iii) **Perception/preprocessing:** target extraction from images/frames (edges/foreground) and sampling exactly N target points;

(iv) **Assignment:** one-to-one drone–target matching (greedy as implemented; Hungarian as an optional optimal alternative);

(v) **(Subtask 3) Motion tracking:** velocity-field tracking via optical flow (implemented as dense Lucas–Kanade in the provided code) or other dense flow methods.

Subtask reports then only need to describe *what changes*: static vs. time-varying targets, and activation of velocity tracking.

2 Notation and state definition

Let N be the number of drones and let $d \in \{2, 3\}$ denote the spatial dimension. For each drone $i \in \{1, \ldots, N\}$:

$$x_i(t) \in \mathbb{R}^d, \qquad v_i(t) \in \mathbb{R}^d.$$

Define the stacked state vector

$$Y(t) = (x_1, v_1, \ldots, x_N, v_N) \in \mathbb{R}^{2dN}.$$

Targets are either static ($T_i \in \mathbb{R}^d$) or time-varying ($T_i(t) \in \mathbb{R}^d$). In Subtask 3, we also use a target velocity estimate $\dot{T}_i(t)$.

3 Shared parameters (physics and numerics)

Table **??** summarizes the core parameters used across subtasks.

Symbol	Code name	Meaning / effect
m	`M`	Mass (inertia). Larger m slows acceleration for the same force.
$v_{\max}$	`V_MAX`	Maximum speed bound via kinematic saturation.
k_p	`K_P`	Spring gain: controls attraction strength to target.
k_d	`K_D`	Damping: dissipates energy, reduces oscillations.
k_{rep}	`K_REP`	Repulsion strength for collision avoidance.
R_{safe}	`R_SAFE`	Safety radius below which repulsion activates.
k_v	`K_V`	Velocity matching gain (Subtask 3 / predictive tracking).
Δt	`DT`	Integration timestep for RK4.

Table 1: Shared parameters across all phases.

Stability intuition (single drone). Ignoring repulsion and assuming a fixed target $T = 0$, the dynamics reduce to

$$\ddot{x} + k_d\dot{x} + k_p x = 0.$$

The damping ratio

$$\zeta = \frac{k_d}{2\sqrt{mk_p}}$$

is selected > 1 (overdamped) for monotonic convergence without oscillations.

4 Core IVP model (backbone used in all subtasks)

All subtasks share the same second-order IVP, with a *velocity-saturated kinematic layer*:

$$\dot{x}_i = v_i \cdot \min\left(1, \frac{v_{\max}}{\|v_i\| + \varepsilon}\right), \tag{1}$$

where $\varepsilon > 0$ prevents division by zero, and dynamics

$$\dot{v}_i = \frac{1}{m}\Big(f_{\text{spring},i} + f_{\text{vm},i} + f_{\text{rep},i} + f_{\text{damp},i}\Big), \tag{2}$$

with force decomposition:

$$f_{\text{spring},i} = k_p\big(T_i(t) - x_i(t)\big), \qquad f_{\text{damp},i} = -k_d v_i(t).$$

4.1 Static targets (Subtasks 1–2)

For static patterns (handwritten name, greeting image), targets do not vary in time:

$$T_i(t) \equiv T_i, \qquad f_{\text{vm},i}(t) \equiv 0.$$

4.2 Dynamic targets and velocity matching (Subtask 3)

For video tracking, the simulator activates velocity matching:

$$f_{\text{vm},i}(t) = k_v\big(\dot{T}_i(t) - v_i(t)\big), \qquad \dot{T}_i(t) \approx \frac{T_i(t_k) - T_i(t_{k-1})}{t_k - t_{k-1}}.$$

This term reduces lag: drones not only chase target positions but also match the target motion trend.

5 Collision avoidance with spatial hashing

Repulsion is compact-support:

$$f_{\text{rep},i}(t) = \sum_{j \neq i} \begin{cases} k_{\text{rep}} \dfrac{x_i - x_j}{\|x_i - x_j\|^3}, & \|x_i - x_j\| < R_{\text{safe}}, \\ 0, & \text{otherwise.} \end{cases}$$

A naive all-pairs evaluation is $O(N^2)$. To improve scalability, the code uses *spatial hashing*:

1. Discretize space into cubic cells of size R_{safe}.
2. Hash each drone position to a cell index $\lfloor x/R_{\text{safe}} \rfloor$.
3. For each drone, check only candidates in its own cell and neighboring cells (3×3 in 2D, 3×3×3 in 3D).

This yields near-linear average behavior for uniform densities.

6 Distance matrix and matching

6.1 Pairwise Euclidean distance matrix

For positions $A \in \mathbb{R}^{N \times d}$ and targets $B \in \mathbb{R}^{N \times d}$, the distance matrix is

$$D_{ij} = \|A_i - B_j\|_2.$$

The implementation uses the identity

$$\|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2a^\top b,$$

to compute all distances efficiently using matrix products.

6.2 Assignment objective

A one-to-one assignment seeks a permutation $\pi \in S_N$ minimizing

$$\pi^\star = \arg\min_{\pi \in S_N} \sum_{i=1}^{N} \|x_i - T_{\pi(i)}\|.$$

Implemented choice: greedy nearest-neighbor (fast, dependency-free).
Optional alternative: Hungarian algorithm (globally optimal but heavier; typically requires SciPy).

7 Numerical integration: classical RK4

The IVP is integrated using classical RK4:

$$Y_{k+1} = Y_k + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

where $k_\ell = f(t_k + c_\ell \Delta t, Y_k + \Delta t \sum_{\nu<\ell} a_{\ell\nu} k_\nu)$. RK4 is selected because it gives high accuracy at practical step sizes and behaves well for damped second-order systems with occasional sharp repulsion impulses.

8 Target extraction (images / frames)

Targets are extracted from an image or frame by:

1. grayscale conversion,
2. mild smoothing (Gaussian blur),
3. edge detection (Canny) and/or foreground extraction (Otsu threshold),
4. sampling exactly N points and scaling into the simulation canvas.

This ensures each subtask produces a target point set compatible with the same physical solver.

9 Optical flow to velocity field $V(x,t)$ (Subtask 3)

The code contains a dense Lucas–Kanade optical flow implementation based on brightness constancy:

$$I(x,y,t) = I(x+u, y+v, t+\Delta t),$$

leading to the linearized constraint

$$I_x u + I_y v + I_t = 0.$$

A local least-squares estimate over a window yields per-pixel flow (u,v), forming a dense velocity field $V(x,t)$. This field is then *sampled at drone positions* via bilinear interpolation.

10 2D to 3D extension

Targets are extracted in 2D (x,y). In 3D mode, a depth coordinate is injected by distributing drones across layers:

$$T_i^{(3D)} = (T_i^{(2D)}, z_i),$$

with z_i chosen from a shuffled layered distribution. This preserves the visible 2D formation while enabling depth-aware rendering.

11 Task–Solution–Math–Code mapping

Requirement / Task	**Implemented solution**	**Mathematical basis**	**Code anchor**
Stable motion to targets	Spring–damper + damping	Eq. (**??**), $\zeta > 1$	`Swarm.step()`
Speed limit	Velocity saturation	Eq. (**??**)	`dx = v * min(...)`
Collision avoidance	Repulsion + spatial hashing	$k_{\text{rep}}(x_i - x_j)/\|x_i - x_j\|^3$	`repulsion()`
Target extraction	Edge/foreground sampling	$\{T_i\} \subset \mathcal{E}$	`extract_points()`
Assignment	Greedy 1–1 matching	$\min_\pi \sum \|x_i - T_{\pi(i)}\|$	`greedy_assignment()`
Dynamic tracking	Velocity matching via $\dot{T}$	$k_v(\dot{T} - v)$	`track_dynamic()`
Time integration	RK4	4th-order IVP solver	`Swarm.step()`

Table 2: Traceability table linking requirements to solution, math, and code.

12 Representative code snippets (optional appendix-style)

12.1 Greedy assignment

```python
def greedy_assignment(positions, targets):
    dist = euclidean_distance_matrix(positions, targets)
    result = np.zeros((len(positions), 2))
    assigned_targets = np.zeros(len(positions), dtype=bool)
    for i in range(len(positions)):
        masked = dist[i].copy()
        masked[assigned_targets] = np.inf
        j = np.argmin(masked)
        result[i] = targets[j]
        assigned_targets[j] = True
    return result
```

12.2 Force model core (static targets)

```python
dx = v * np.minimum(1.0, V_MAX / vnorm)  # velocity saturation
spring_force = K_P * (self.tgt - p)
rep = self.repulsion()
damping = K_D * v
dv = (spring_force + rep - damping) / M
```

13 AI usage disclosure

Parts of the report structure and wording were refined with assistance from a large language model; all mathematical definitions, parameter choices, and code behavior were verified against the submitted implementation.