

# GenAI Usage Documentation

## Tool Used

I used **ChatGPT-5.2 Thinking** as a GenAI assistant to help generate and refine Python code for a Boids flocking simulation.

## GenAI Prompts and Outputs for All Modules

### Prompt 1: Discuss the Implementation Plan

I'm implementing boids in Python, but I'm not a coding expert.

Can you propose a simple implementation plan for me:

- How should I store boid states (position/velocity)?
- What functions do I need (rules, update step, boundaries)?
- What parameters should be global?

Please keep it simple and explain in plain English.

### Prompt 1 Output Summary

**State Representation.** I will store boid states using NumPy arrays for simplicity and efficiency:

- `positions`  $\in R^{N \times 2}$ , where `positions[i]` =  $(x_i, y_i)$
- `velocities`  $\in R^{N \times 2}$ , where `velocities[i]` =  $(v_{x,i}, v_{y,i})$
- optionally `accelerations`  $\in R^{N \times 2}$  to store steering forces per step

**Main Functions.** Each simulation iteration follows the idea *sense*  $\rightarrow$  *steer*  $\rightarrow$  *move*  $\rightarrow$  *handle boundaries*. The planned functions are:

- Vector helpers: `normalize_or_zero(v)` and `limit_magnitude(v, max_val)`
- Neighbor search: `get_neighbors(i, positions, radius)`
- Rule steering (each returns a 2D steering vector): `steer_separation`, `steer_alignment`, `steer_cohesion`
- Rule combination: `compute_acceleration(i, positions, velocities, params)`
- Simulation update: `step(positions, velocities, params)` using a two-phase update (compute all accelerations first, then update all boids)
- Boundary handling: `apply_boundary(pos, vel, params)` supporting "wrap" and "reflect" modes

**Global Parameters.** All parameters will be stored in a single container (dictionary or small dataclass), including:

- Simulation/world: `N`, `width`, `height`,  `$\Delta t$`
- Neighborhood radii: `perception_radius`, `sep_radius`
- Rule weights:  `$w_{sep}$` ,  `$w_{align}$` ,  `$w_{coh}$`
- Stability limits: `max_speed`, `max_force`
- Boundaries: `boundary_mode`  $\in \{\text{wrap}, \text{reflect}\}$

**Iteration Flow.** In each time step, for each boid: find neighbors, compute separation/alignment/cohesion steering vectors, combine them with weights to obtain acceleration, update velocity (clamp speed), update position, and finally apply the chosen boundary rule.

### Prompt 2: Minimal Code Skeleton

Based on your plan, write a minimal Python skeleton. I want:

- initialization of positions/velocities
- a simulation loop `step()` that updates positions
- boundary handling (wrap and reflect)

No flocking rules yet. Use numpy.

Also show a tiny smoke test that runs 10 steps.

### Prompt 2 Output Summary

**Code.** The GenAI provided a minimal Boids simulation skeleton in Python using NumPy, where boid states are stored as position and velocity arrays of shape  $(N, 2)$ . Each call to `step()` updates positions using constant velocities and then applies either wrap-around or reflect boundary handling to keep boids within the world. I verified the implementation with a small smoke test that ran 10 steps for both boundary modes and printed “Smoke test passed (wrap and reflect).”

### Prompt 3: Add the Three Rules

Now can you add separation, alignment, and cohesion to the code? Please keep the code readable and add comments. Also tell me what parameters I should tune (radii, weights, max speed).

### Prompt 3 Output Summary

**Code.** The GenAI extended the minimal boids skeleton by adding the three flocking rules (separation, alignment, and cohesion) implemented as steering vectors computed from nearby neighbors within a perception radius. In each time step, the code first computes all boids’ accelerations from the current state (two-phase update to avoid update-order bias), then updates velocities (with speed clamping), updates positions, and finally applies either wrap-around or reflect boundary handling. A smoke test running 10 steps verifies numerical stability (no NaNs) and confirms that all boid speeds remain below the specified maximum speed.

### Prompt 4: Visualization

Can you help me visualize it in Python? I prefer matplotlib animation. Just show a simple moving plot of the boids.

### Prompt 4 Output Summary

**Code.** To visualize the boids in a Jupyter notebook, the GenAI first provided a standard Matplotlib animation approach (e.g., `FuncAnimation`) and then tried a manual “live update” loop using `plt.pause()`. However, in Jupyter these methods may fail when the plotting backend is non-interactive (commonly the default inline backend), so no motion is displayed. To resolve this, the GenAI was asked to proposed a notebook-friendly solution: rendering the animation inline by converting it to JavaScript/HTML using `HTML(ani.to_jshtml())`, which reliably displays the moving boids directly inside the notebook. It turned out that converting the animation to HTML bypassed the backend limitation and successfully displayed the moving boids inside the notebook.

### Prompt 5: Quick Correctness Checks

Can you suggest a few simple tests or checks to make sure my boids code is correct? Nothing fancy, just quick checks I can run.

## Prompt 5 Output Summary

To quickly sanity-check the boids implementation, I used several lightweight tests. First, I ran the simulation for many steps and asserted that positions and velocities remain finite (no NaNs/Infs) and that the maximum speed never exceeds the specified `max_speed`. Second, I verified boundary handling by checking that boids remain within the valid coordinate ranges under both wrap-around and reflect modes. Third, I performed simple rule “toggle” checks by setting weights so that only one rule is active at a time (separation-only should spread boids out, cohesion-only should clump them, alignment-only should reduce heading differences for nearby boids, and all three together should produce stable flocking). Finally, I checked important edge cases such as a boid with no neighbors (should yield near-zero steering without numerical issues) and a symmetry setup where two equally spaced neighbors should produce approximately canceling separation forces, which helps detect sign or direction bugs.

## Reflection on the GenAI Process

Using GenAI for the boids simulation was most effective when I treated it as a collaborator rather than a code generator. The prompts that worked best were structured and goal-oriented: I first asked GenAI to discuss an implementation plan (how to store state, what functions are needed, and what parameters to expose), and only then requested code module-by-module. This planning step was new to me—previously I would brainstorm alone and directly start coding—but it made the workflow smoother because GenAI maintained a consistent understanding across modules (state representation, update order, and boundary handling). In contrast, the least effective prompts were vague requests such as “make it animate” without specifying constraints or context; they tended to produce code that was technically reasonable but not compatible with my actual setup.

GenAI’s main mistakes were not conceptual errors in boids rules, but practical implementation assumptions. The best example was the visualization: the initial Matplotlib animation did not display in my Jupyter notebook environment. I caught the issue through direct execution (no animation shown) and corrected it by explicitly stating my environment and trying alternatives. Switching to an HTML-based rendering approach (converting the animation to JavaScript/HTML) ultimately worked reliably in Jupyter. This taught me that prompts should include the programming environment (e.g., Jupyter vs. script) because it affects what solutions are feasible.

The requirement to understand the algorithm before prompting clearly changed how I worked: with the conceptual model in place, I could evaluate GenAI outputs critically and request targeted fixes instead of blindly accepting code. Next time, I would specify the environment earlier, keep prompts concise but organized, and add simple sanity checks (NaNs, speed limits, boundary constraints) immediately after each module.

## Tier 1: Spatial Optimization with KDTree (scipy.spatial)

### Implementation Plan (Before Coding)

**Plan Summary.** I will keep the simulation logic unchanged (rules, update order, boundaries), and only replace the neighbor-identification step. Each frame, I will build a KDTree from the current positions and use radius queries to retrieve neighbor lists efficiently.

#### Key Design Decisions.

- **Tree rebuild frequency:** rebuild the KDTree once per simulation step (positions change each frame).
- **Query type:** use a radius query to match the “neighbors within perception radius” logic.
- **Correctness validation:** for a fixed random seed and small  $N$ , compare naive vs. KDTree neighbor sets (or the resulting accelerations) and confirm they match.
- **Benchmarking:** time naive and KDTree versions for  $N \in \{50, 200, 500\}$  using the same parameters and number of steps; report mean time per step.

### Module 1: Baseline Timing (Naive Neighbor Search)

#### Prompts Used.

- “I already have a working boids simulation with naive neighbor search. How can I benchmark the runtime fairly for different  $N$  (50/200/500) in Python? Please suggest a simple timing script and what to time (neighbor finding vs. full step). Keep it beginner-friendly.”

**GenAI Output (Summary).** GenAI suggested timing the simulation using a fixed number of warm-up steps and measuring average time per step across multiple runs. It also suggested keeping the random seed fixed for comparability.

#### My Modifications.

- I timed *full step runtime* (more realistic) and also timed *neighbor search only* to isolate the optimization.
- I standardized parameters and seeds across naive vs. KDTree comparisons.

#### Tests / Verification.

- Verified that timing code runs without errors and returns stable estimates (repeatable within small noise).

### Module 2: KDTree Refactor (Optimized Neighbor Search)

#### Prompts Used.

- “Please help me refactor my boids neighbor search to use `scipy.spatial.KDTree` (or `cKDTree`). I currently have `get_neighbors(i, positions, radius)` using an  $O(N^2)$  distance check. I want an equivalent function using KDTree radius queries. Please keep the code readable and explain the key lines.”
- “Important: positions update every frame. Where should I build/update the tree so it is efficient? (I am worried about rebuilding too often.)”

**GenAI Output (Summary).** GenAI provided a KDTree-based neighbor query approach using a radius query function (e.g., `query_ball_point`), and explained that the KDTree should be constructed once per frame from the current positions and reused for all boids that frame.

**My Modifications.** During review and testing, I checked for the following issues and corrected them when necessary:

- **Inefficient rebuilding:** avoided rebuilding the tree inside the per-boid loop; rebuilt only once per frame.
- **Self-neighbor bug:** ensured the returned neighbor list excludes the boid itself.
- **Radius mismatch:** confirmed the KDTree query uses the same radius definition as the naive implementation.
- **Data types/shapes:** ensured positions are a float array of shape  $(N, 2)$  before building the tree.

**Tests / Verification.**

- Smoke test: run 10–50 steps with KDTree enabled and check no NaNs/Infs and speed clamping still holds.
- Sanity check: neighbor counts are reasonable (not all boids are neighbors unless radius is huge).

### Module 3: Correctness Check (Naive vs. KDTree at Small Scale)

**Prompts Used.**

- “How can I demonstrate that KDTree neighbor search produces identical behavior to my naive implementation for small  $N$ ? Please propose a simple test comparing either (a) neighbor sets or (b) the resulting acceleration vectors, using a fixed random seed.”

**GenAI Output (Summary).** GenAI proposed running one or a few steps with the same initial conditions and comparing outputs from naive vs. KDTree. It suggested using assertions with a small tolerance for floating-point differences.

**My Modifications.**

- I fixed the random seed and compared either the neighbor indices (order-insensitive) or the final accelerations per boid.
- If mismatch occurred, I traced it back to boundary handling or self-neighbor filtering and corrected it.

**Tests / Verification.**

- Correctness test (example): for  $N = 50$ , same seed and parameters, the maximum difference between naive and KDTree accelerations satisfies  $\max_i \|\Delta \mathbf{a}_i\| < \varepsilon$  (e.g.,  $\varepsilon = 10^{-9}$ ), or the neighbor sets match exactly.

### Module 4: Performance Benchmark + Table/Figure

**Prompts Used.**

- “Please help me write a short benchmark: compare runtime (naive vs. KDTree) for  $N = 50, 200, 500$  boids. I want average time per step over multiple steps, and a table or plot of time vs.  $N$ . Keep it simple for a report.”

**GenAI Output (Summary).** GenAI suggested timing loops for each  $N$ , computing average time per step, and presenting the results in a table and/or a plot.

### My Modifications.

- I ensured both versions use identical parameters, step counts, and seeds.
- I included a brief warm-up before timing to avoid startup overhead dominating results.

### Results.

Flock size $N$	Naive time/step (ms)	KDTree time/step (ms)	Speedup ( $\times$ )
50	$1.917 \pm 0.029$	$1.882 \pm 0.035$	1.02x
200	$10.763 \pm 0.846$	$10.441 \pm 0.970$	1.03x
500	$35.877 \pm 0.313$	$33.077 \pm 0.788$	1.08x

Table 1: Timing comparison between naive  $O(N^2)$  neighbor search and KDTree-based neighbor search.

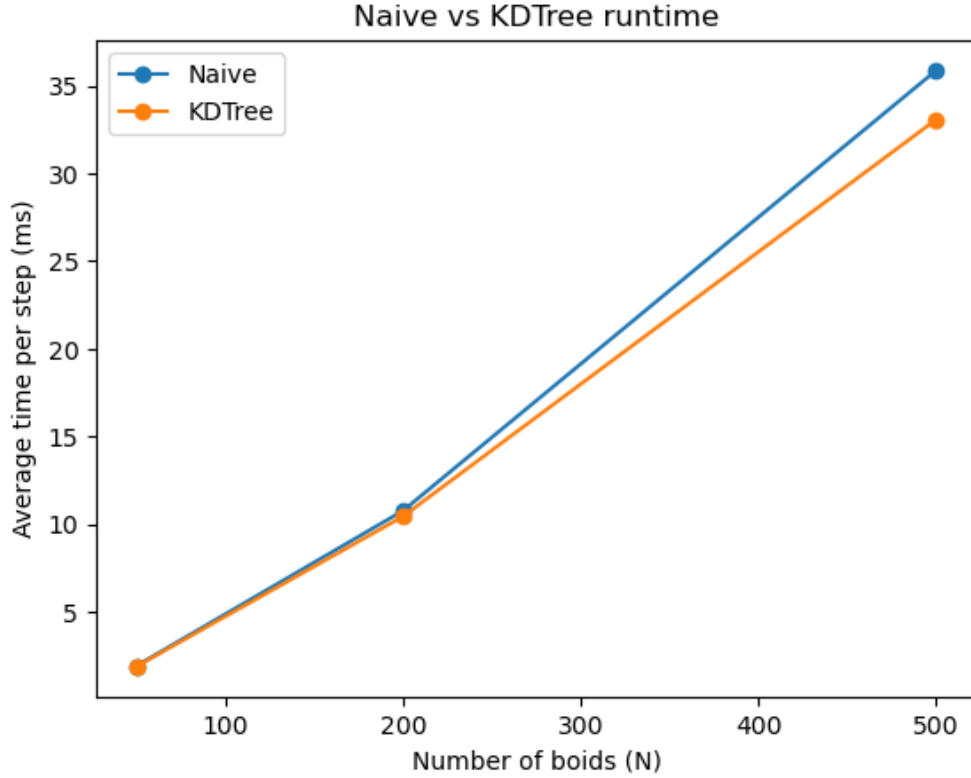


Figure 1: Runtime comparison (naive vs. KDTree).

### Short Reflection on Optimization Prompting

The most effective prompts were those that stated the goal (KDTree refactor), clarified what must remain identical (behavior at small scale), and explicitly mentioned boundary mode (wrap vs. reflect). The main difficulties were edge cases: ensuring the tree is rebuilt once per frame (not per boid), excluding self from neighbor lists, and matching wrap-around distance in KDTree queries. These issues were caught by small-scale equivalence tests and corrected before benchmarking.

## Tier 2: Predator Avoidance

### Implementation Plan

**Plan Summary.** I will extend the simulation by adding one additional agent (the predator) and one additional steering term (avoidance). The original three boid rules (separation, alignment, cohesion) will remain unchanged, but the final acceleration will include a predator-avoidance component with a higher weight or a priority mechanism.

#### Core Design Choices.

- **State representation:** predator has **position** and **velocity** (same 2D vectors as boids).
- **Boid update:** compute boid acceleration as

$$\mathbf{a}_i = w_{\text{sep}}\mathbf{a}_i^{\text{sep}} + w_{\text{ali}}\mathbf{a}_i^{\text{ali}} + w_{\text{coh}}\mathbf{a}_i^{\text{coh}} + w_{\text{pred}}\mathbf{a}_i^{\text{pred}},$$

where  $\mathbf{a}_i^{\text{pred}}$  steers away from the predator.

- **Priority handling:** if the predator is within a danger radius, scale up avoidance or reduce cohesion (override behavior).
- **Predator update rule:** predator steers toward (a) flock center or (b) nearest boid, then updates position/velocity with its own speed and force limits.
- **Experiment design:** run two scenarios with the same initial seed: (i) predator absent, (ii) predator present; then remove predator mid-run to show recovery.

### Module 1: Predator Class and Integration Point

#### Prompts Used.

- “I want to add a predator agent to my existing boids simulation. Please propose a minimal **Predator** class with **position** and **velocity** (NumPy arrays), and a **step()** method that updates it each frame. Keep it consistent with my boid update style (max speed, max force, dt, boundary mode).”
- “Where should the predator update happen relative to boid updates (before/after)? Please recommend a simple, stable order.”

**GenAI Output (Summary).** GenAI suggested representing the predator similarly to a boid (position/velocity vectors), updating it once per frame, and applying the same boundary function. It recommended updating the predator either before boids (so boids react to the newest predator state) or using a two-phase update to avoid bias.

#### My Modifications

- I updated the predator once per frame and ensured boundary handling matches the simulation mode.
- I chose a consistent update order: compute predator steering → update predator → compute boid accelerations using current predator state.

#### Tests / Verification.

- Smoke test: predator can be instantiated, updated for 10 steps, and remains within valid bounds (wrap/reflect).
- Numerical test: predator position/velocity remain finite; predator speed is clamped.

## Module 2: Predator Avoidance Rule (Fourth Rule)

### Prompts Used.

- “Please implement a fourth steering rule for boids: predator avoidance. Each boid should steer away from the predator when the predator is within an avoidance radius. The avoidance should have high priority (it may override cohesion). Return an acceleration/steering vector with magnitude limited by `max_force`. Add clear comments.”
- “What is a simple way to implement priority? For example, if predator distance  $< r_{\text{danger}}$ , reduce cohesion weight or directly add a strong avoidance term. Please suggest a beginner-friendly approach.”

**GenAI Output (Summary).** GenAI proposed computing a vector from predator to boid (repulsion direction) and converting it to a steering vector similar to separation (desired direction minus current velocity). It also suggested a “danger radius” mechanism where avoidance weight increases sharply and cohesion is reduced when the predator is close.

### My Modifications / Corrections (Typical Failure Modes).

- **Predator too weak:** if boids ignore predator, I increased  $w_{\text{pred}}$  and/or expanded the avoidance radius.
- **Overreaction/jitter:** if boids explode or oscillate, I reduced  $w_{\text{pred}}$ , lowered `max_force`, or decreased `dt`.
- **Priority conflict:** if cohesion still dominates near predator, I explicitly reduced cohesion when within danger radius (override).

### Tests / Verification.

- Edge case: if predator is far away, avoidance acceleration is near zero.
- Sanity check: if predator is placed close to the flock, boids should rapidly increase spacing and move away.

## Module 3: Predator Movement Rule (Chase/Track Behavior)

### Prompts Used.

- “Please implement predator movement toward (a) the center of the flock or (b) the nearest boid. Keep it simple: compute a desired direction and apply a steering force limited by `max_force` and speed limited by `max_speed`. Which approach is more stable for demonstration, and why?”

**GenAI Output (Summary).** GenAI provided two options: chasing the nearest boid (more dynamic and disruptive) or moving toward the flock center (more stable and predictable). It recommended the center-chasing variant as easier to debug and the nearest-chasing variant as more visually dramatic.

### My Modifications / Corrections.

- I chose one predator mode for the main report (center or nearest) and fixed it for fair comparisons.
- If behavior was too chaotic, I lowered predator max speed or max force, or used center-chasing instead of nearest-chasing.

### Tests / Verification.

- Predator should consistently move toward its target (center/nearest) under both boundary modes.



## Module 4: Demonstration Scenario (Disruption and Recovery)

### Prompts Used.

- “Help me design a simple demonstration: run the simulation without predator for some steps, then enable predator and show the flock is disrupted, then remove predator and show recovery. Please suggest a simple schedule (e.g., predator off for 200 frames, on for 200, off for 200) and what to visualize.”

**GenAI Output (Summary).** GenAI suggested a three-phase schedule (no predator → predator present → predator removed) and visualizing boid positions, optionally drawing the predator with a different marker/color.

### My Modifications / Corrections.

- I ensured the animation clearly distinguishes predator from boids (marker shape/size).
- I kept random seed fixed across runs for reproducibility.

### Tests / Verification.

- Behavioral check: flock cohesion decreases or splits when predator is present; flock re-forms after predator removal.

### Short Reflection (Tier 2)

The most effective prompts specified integration constraints: preserve the original three rules, add a fourth rule with explicit priority near the predator, and keep predator motion simple and testable. The main debugging steps involved adjusting the avoidance strength and radii when the predator had no visible effect, and reducing forces/time step when the system became unstable.