Final Project

Swarm Navigation

1.1 Objectives

The purpose of this project is to design, implement, and test a multi-robot navigation function. In this project, you will:

- Design a navigation function U (or equivalent control law) that guides multiple robots to their targets.
- Implement the gradient of your function in Python to control the robots.
- Ensure your controller handles robot-robot, robot-obstacle, and robot-boundary collisions according to the specified semantics.
- Test your implementation across multiple configurations and target permutations.
- Write a formal report analyzing your design choices, test plan, and results.

1.2 Reference

This project builds on concepts from potential fields and navigation functions (cf. Modern Robotics, Ch. 10.6). You may refer to class lecture notes and standard robotics textbooks on motion planning.

1.3 Introduction

1.3.1 Navigation functions

We consider feedback-based motion planning in a fully actuated setting. Let the configuration space be a simple convex domain $D \subset \mathbb{R}^2$ (here: the *unit disk*), with a target $p \in D$. In the unobstructed case, a *target* (Lyapunov) function f with anti-gradient field

$$v(x) \;=\; -\nabla f(x)$$

can be chosen so that (a) v points inward on ∂D and (b) all trajectories converge to p. One may view f as a potential energy minimized at the goal. The anti-gradient of this function creates a vector field where all paths converge to the target, as shown in Figure 1.1.

1.3.2 Obstacles and barriers

When obstacles $\{\mathcal{O}_\alpha\}_{\alpha \in A} \subset D$ are present, we define the free space $\mathcal{C}_{\text{free}} = D \setminus \bigcup_\alpha \mathcal{O}_\alpha$ and augment f by a *barrier* g that grows near obstacle boundaries. A standard device (cf. Koditschek–Rimon) is the *navigation function*

$$U(x) \;=\; f(x) \;+\; \epsilon\, g(x), \qquad \epsilon > 0,$$

whose anti-gradient $v_b(x) = -\nabla U(x)$ attracts toward the goal far from obstacles while repelling near them.
This is illustrated in the figures below. Figure 1.2 shows an example barrier function that explodes near the obstacle. Figure 1.3 and Figure 1.4 show the resulting anti-gradient vector field for a strong barrier ($\epsilon = 1.5$) and a weaker barrier ($\epsilon = 0.5$), respectively.

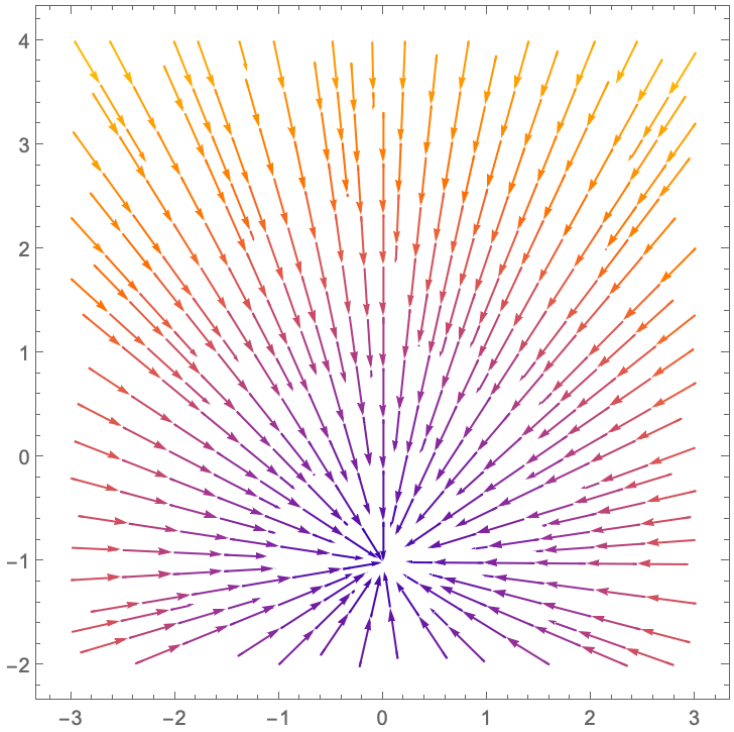

Figure 1.1: An example target function $f = x^2+(y+1)^2$. The anti-gradient vector field (shown) converges to the target at $p = (0, -1)$.

1.3.3 Multi-robot setting and collision semantics

We study $n = 5$ mobile robots moving in the unit disk. Each robot is modeled as a disk of *radius* $r > 0$ (fixed for grading). Collision rules:

- **Boundary (unit circle):** *center-only* rule — a step is invalid if $\|x_k\| > 1$.
- **Robot–obstacle:** *outer-edge* rule — collision if $\|x_k - c_m\| \le R_m + r$.
- **Robot–robot:** *outer-edge* rule — collision if $\|x_i - x_j\| \le 2r$.

Robots start on the unit circle at angles $\theta_k = k\pi/n$ $(k = 1, \ldots, n)$ and must reach assigned targets at angles $\theta'_k = \sigma_k \pi/n + \pi$, where σ is a permutation of $\{1, \ldots, n\}$. There are $n! = 120$ target permutations per configuration. Figure 1.5 shows an example of the starting configuration.

1.4 Tasks

1.4.1 Design of the Navigation Field

1. **Define a navigation function U:** Propose $U : \mathbb{R}^{2n} \to \mathbb{R}$ (or an equivalent control law) whose anti-gradient induces collision-free motion toward the assigned targets. Typical decompositions include

$$U(X) \;=\; U_{\text{goal}}(X) \;+\; \lambda_{\text{obs}}\, U_{\text{obs}}(X) \;+\; \lambda_{\text{pair}}\, U_{\text{pair}}(X) \;\; (+ \text{ boundary term if used}),$$

   with designable weights/exponents.

2. **Gradient for control:** Derive closed-form expressions for $-\nabla_{(x_k, y_k)} U$ for each robot k, suitable for step-by-step simulation. You may include practical regularizers (e.g., ε in denominators) and the envelope will cap speed to $v_{\max}$.

3. **Respect collision semantics:** Ensure your design encodes the rules above (center-only boundary; outer-edge for obstacles and pairwise separation).

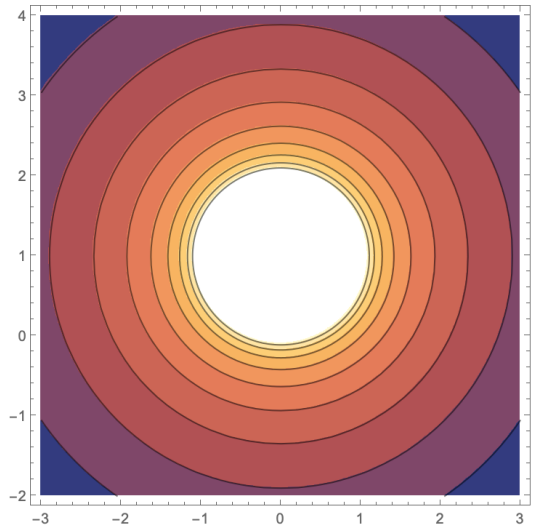

Figure 1.2: Contour plot of an example barrier function $g = -\log(x^2+(y-1)^2-1)$, which explodes near the obstacle boundary.

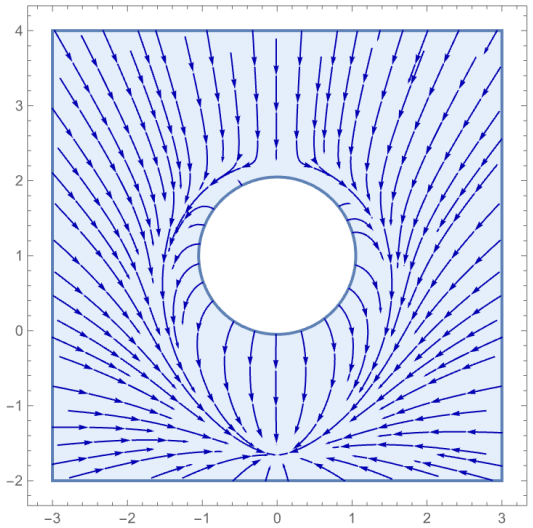

Figure 1.3: The combined navigation function $U = f + \epsilon g$ with $\epsilon = 1.5$. The barrier strongly repels the flow.

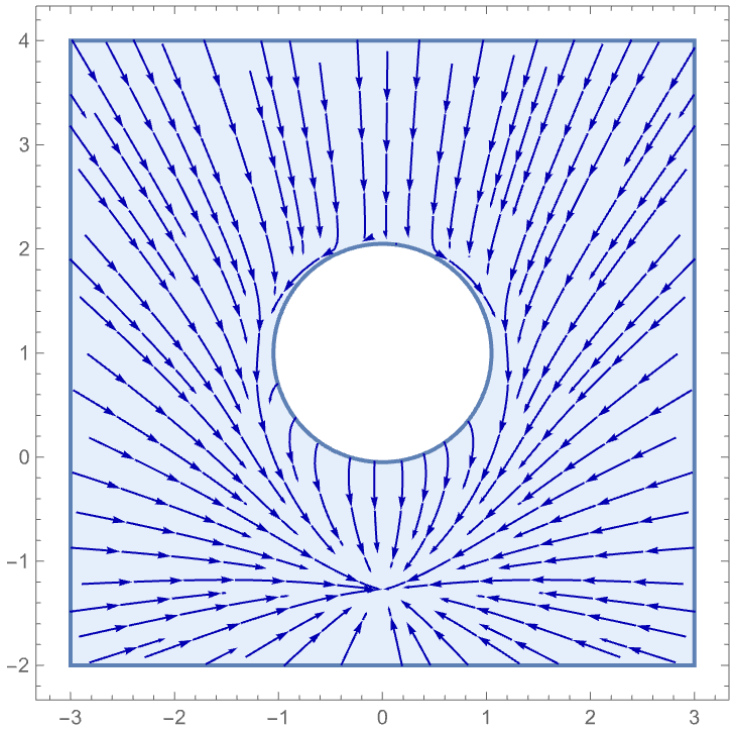

Figure 1.4: The navigation function $U = f + \epsilon g$ is similar to one above, but with a weaker barrier, $\epsilon = 0.5$. The flow passes closer to the obstacle.

1.4.2 Code Implementation

1. **Student API:** Implement your field in the provided function

   `compute_gradients(state, targets, obstacles, r)` → `(n,2) array`

   returning the velocity for each robot at the current state. The simulator handles step integration, speed capping, and validity checks.

2. **Targets & permutations:** Your code will be evaluated over all $n! = 120$ permutations per configuration; no changes are needed to your API to handle different σ.

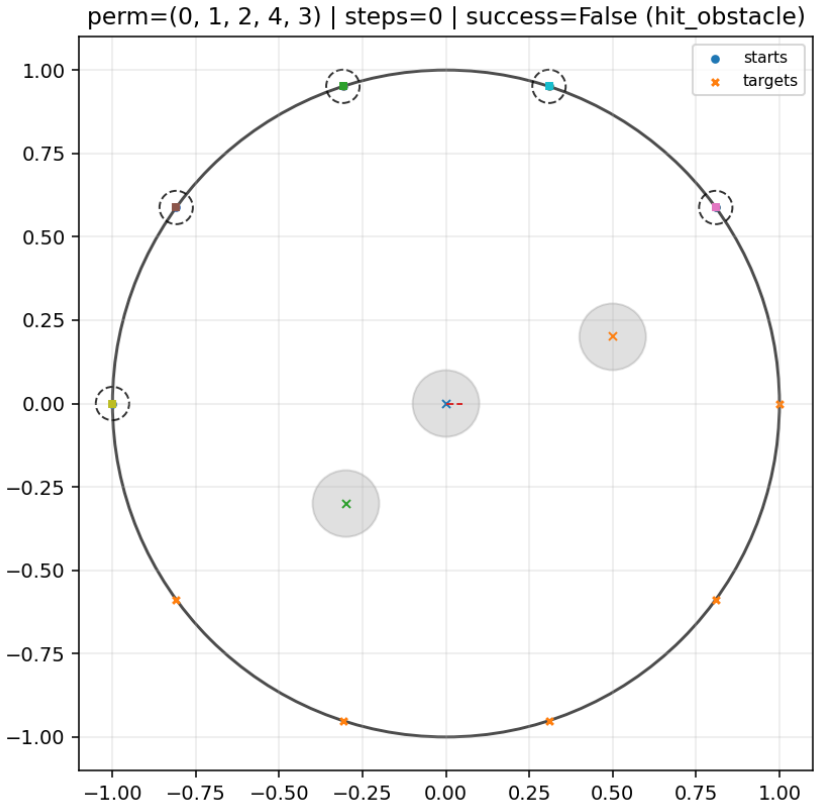

Figure 1.5: The project's C0 (public) configuration, showing the 5 robots (dots) in their starting positions on the unit circle, their 5 targets (crosses), and the 3 circular obstacles. (Image from `visualize.py`)

1.4.3 Notes

For local exploration in your report you may experiment on the public configuration (e.g., small obstacle shifts or small variations of r), but grading runs use staff configurations with fixed r and perturbed obstacle *centers*. Only your `my_nav_fn.py` is used for grading.

1.5 Procedure

1.5.1 Project Structure

The project is structured so that you only need to focus on the navigation logic, while the simulation and visualization are handled by the provided "envelope" code. The final project directory is organized as follows:

- `student_code/my_nav_fn.py`: **This is the main file you will edit.**
- `run_all.py`: Main script to run all simulations.
- `visualize.py`: Main script to render results from simulations.
- `envelope/`: Contains the core simulator code (`simulate.py`, `geometry.py`, `config.py`).
- `results/`: This directory will be created by `run_all.py` to store all simulation output.

1.5.2 Implementation Workflow

Your main task is to implement your navigation function inside `student_code/my_nav_fn.py`. Your workflow will follow a simple edit-run-visualize cycle.

1. Implement Your Navigation Function

Open `student_code/my_nav_fn.py`. Your entire implementation will be inside the `compute_gradients` function. A naive placeholder is provided, which you must replace with your own logic. The function signature and requirements are as follows:

```python
import numpy as np

def compute_gradients(state, targets, obstacles, r):
    """
    Compute a velocity field (interpreted as -grad U).

    Inputs:
      state: (n, 2) ndarray. Current xy positions for n robots.
      targets: (n, 2) ndarray. Target xy positions for n robots.
      obstacles: list of dicts. e.g., [{"center": (x, y), "radius": R}]
      r: float. REQUIRED minimum pairwise robot distance.
         Simulation fails if any pair distance is <= r.
         Use this to shape your inter-robot repulsion.

    Returns:
      grads: (n, 2) ndarray. Velocity vectors for each robot.
             The envelope will cap speed and step the system.
             IMPORTANT: Must be finite (no NaN/Inf).
    """

    # ============================================================
    # TODO: Replace this naive baseline with YOUR navigation field.
    # ============================================================

    # --- Naive placeholder baseline (will often fail): straight to target.
    V = targets - state
    norms = np.linalg.norm(V, axis=1, keepdims=True) + 1e-12
    return V / norms  # unit direction toward target
```

Listing 1.1: The `compute_gradients` function in `my_nav_fn.py`.

2. Run the Simulation

Once you have an implementation you wish to test, run the full simulation from the main project directory:

```
$ python3 run_all.py
```

This script will import your `compute_gradients` function, run it for all 120 target permutations, and save all results into the `results/` directory.

3. Visualize and Analyze Results

To see the paths your robots took, run the visualization script:

```
$ python3 visualize.py
```

This script reads the `.json` files from `results/traces/` and generates `.png` images for each run in `results/figures/`.

4. Iterate

Check the generated figures and the `results/summary.csv` file to understand your successes and failures. Tune your navigation function in `my_nav_fn.py` and repeat the cycle.

1.5.3 Simulation Parameters

While you will not be graded on changing parameters, you are free to experiment by editing `envelope/config.py`. This file controls:

- **Robot radius r:** Affects collision checks (robot–robot uses $2r$; robot–obstacle uses $R_{\text{obs}} + r$).
- **Obstacles:** The list of obstacles, their centers, and their radii.
- **Simulation knobs:** `dt`, `vmax`, `tol` (tolerance for reaching target), and `max_steps`.

1.5.4 Understanding the Output

The `run_all.py` script generates three types of output in the `results/` folder. An example of the graphical output from `visualize.py` is shown in Figure 1.6.

- `summary.csv`: The most important high-level file. Shows success/fail, reason for failure (if any), and step count for every permutation.
- `traces/perm_###.json`: Contains the full, detailed trajectory data for each run.
- `figures/perm_###.png`: The PNG images, like the one below, generated by `visualize.py`, showing the paths, targets, and final robot positions.

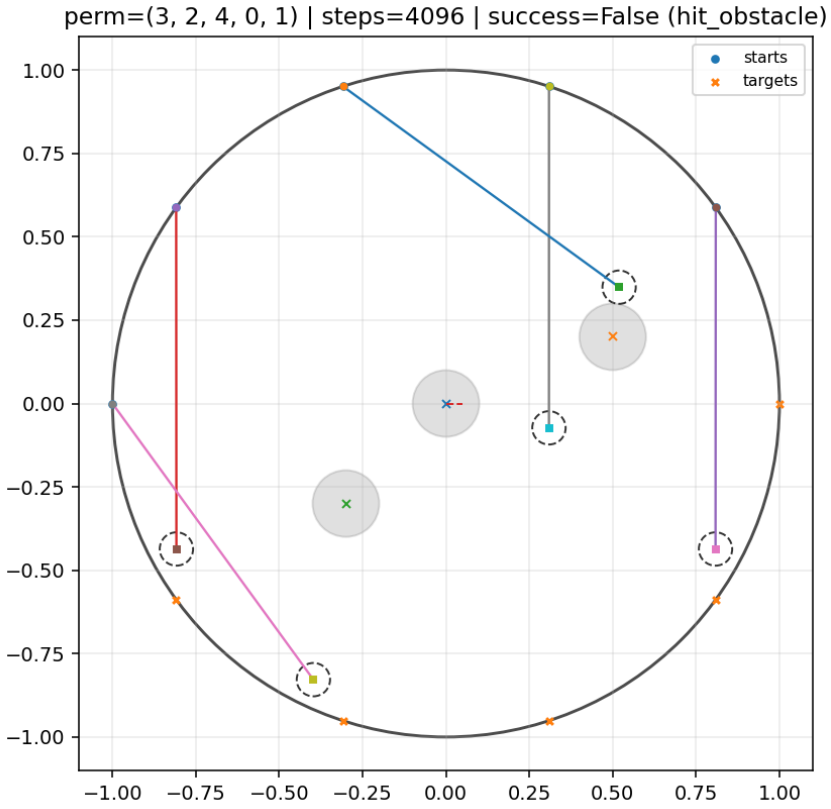

Figure 1.6: An example of a failed run (permutation 88) from `visualize.py`. The plot shows the path of each robot (colored lines), and the simulation failed because a robot hit an obstacle.

1.6 Report

You should submit a lab report using the guidelines given in Appendix B. Your report **must** include the following:

- A clearly written derivation of your navigation function U and its gradient.
- A discussion of your test plan and the rationale for it.
- For the public configuration (C0), include:
  - The overall success rate across all 120 permutations.
  - The average steps to completion for successful runs.
  - At least two trajectory plots: one showing a typical successful run and one showing an interesting or challenging case (e.g., a failure, a near-miss, or a complex path).

- A discussion of your results from testing on your own other configurations.
- A discussion of failure modes, trade-offs, and lessons learned.

1.7 Submission

There are two submissions.

- For the code submit your `student_code/my_nav_fn.py` file to the final project code assignment.
- For the report submit it to the final project report Gradescope assignment.

Submission deadlines are on Gradescope.

1.8 Grading

Your final project grade is divided into two main components, each worth 50%:

- **Code Performance (50%)**
- **Project Report (50%)**

1.8.1 Code Performance (50%)

This portion of your grade is a competition. Your submitted `my_nav_fn.py` will be run on the public configuration (C0) and three hidden configurations (C1-C3).
For each configuration, we will run all 120 target permutations and compute a performance metric based on a combination of **accuracy** (success rate, i.e., avoiding all collisions and boundary violations) and **speed** (efficiency, e.g., average steps to completion).
Your final performance score will be based on your metric averaged across all four test cases. This score will then be compared relative to the performance of all other students in the class to determine your grade for this section.

1.8.2 Project Report (50%)

This report is a significant component of your final grade. Unlike previous labs, this report constitutes 50% of your project grade, and the grading will be stricter as a result.
Your report will be graded based on its clarity, completeness, and the depth of your analysis, using a detailed rubric on GradeScope. To earn full credit, you must thoroughly address all topics listed in the **Report** section and follow the detailed structure guide in **Appendix B**. We are looking for a professional-quality document that clearly explains your design, justifies your choices, analyzes your results, and discusses what you learned.

Appendix A

Using Python Locally

A.1 Overview

This appendix describes how to run the project locally with Python (no ROS required). You will install Python 3.10+, create an isolated virtual environment, install dependencies, and run the provided scripts to simulate and visualize results.

A.2 Install Python (3.10+)

- Download from the official site: `https://www.python.org/downloads/`
- **Windows:** during install, check "Add Python to PATH."
- Verify the installation:

```
# macOS / Linux
$ python3 --version

# Windows (CMD or PowerShell)
> python --version
```

A.3 Set Up a Virtual Environment (recommended)

In the project folder:

- Create:

```
# macOS / Linux
$ python3 -m venv .venv

# Windows
> python -m venv .venv
```

- Activate:

```
# macOS / Linux (bash/zsh)
$ source .venv/bin/activate

# Windows (PowerShell)
> .\.venv\Scripts\Activate.ps1

# Windows (CMD)
> .venv\Scripts\activate
```

A.4 Install Dependencies

- With the virtual environment **activated**:

```
# Either form is fine; prefer the python -m form if PATH is ambiguous
(.venv) $ python -m pip install -r requirements.txt
# or
(.venv) $ pip install -r requirements.txt
```

A.5 Run the Project

- Simulate all permutations (writes summaries and traces):

```
(.venv) $ python run_all.py
```

- Render figures from saved results (no simulation):

```
(.venv) $ python visualize.py
```

A.6 Outputs (Where to Look)

- `results/summary.csv` — aggregate outcomes by permutation (success/fail, reason, steps).
- `results/runs/perm_###.json` — per-permutation run summaries.
- `results/traces/perm_###.json` — full trajectory data (0-based index).
- `results/figures/perm_###.png` — rendered plots (environment, paths, targets, final r-circles).

A.7 Helpful References

- Virtual environments: `https://docs.python.org/3/library/venv.html`
- VS Code + Python guide: `https://code.visualstudio.com/docs/python/python-tutorial`

Appendix B

Report Writing Guide

B.1 Overview

This appendix provides a detailed guide on the expected structure and content for your final report. The goal is to explain what you built and why, the problems you faced and how you addressed them, how you tested, and what you learned. Use the standard IMRaD structure (Introduction, Method, Results, Discussion, Conclusion).

B.2 Expected Structure

1. **Overview / Introduction (0.5–1 page)**
   - Briefly describe the task in your own words (multi-robot navigation with obstacles in a unit disk; robots are disks of radius r; collision rules).
   - State your goals (e.g., minimize failures, reduce steps, remain stable near obstacles).
   - Identify your design philosophy (e.g., potential field with tuned barriers; smoothing; robustness over aggressive speed).
2. **Method**
   - Navigation function U (or equivalent control law): define key terms (target attraction, obstacle barrier, pairwise separation, boundary handling).
   - Why these terms/weights? Provide a brief rationale for each (what behavior it induces; expected trade-offs).
   - Gradient/velocity computation: show the essential math (not every line), plus any stabilizers (small ε terms, clamps, saturation at $v_{\max}$).
   - Implementation details that mattered: step-size choices (Δt), handling near singularities, deadlock-avoidance tricks, line search or damping if used.
3. **Results**
   Report what you tested, why you chose those tests, and how your controller performed.
   - **Test plan & rationale.** Describe your chosen evaluation setups: which configurations you ran (robot radius r, obstacle layouts, other factors), and why these tests are informative (stress specific behaviors, tight passages, symmetry breaking, etc.).
   - **Metrics.** For each test set, report at least:
     - Success rate (fraction of permutations completed without collisions/violations)
     - Average steps to completion (mean ± std over successful runs)

     You may include additional metrics if helpful (e.g., max/min steps, near-miss counts, path length).
   - **Representative figures.** Include 1–3 trajectory plots that illustrate typical behavior, a challenging case, and (if applicable) a failure mode. Mark robot-radius r circles at final positions and annotate notable events (near-misses, stalls, boundary brushes).
   - **Comparative insight (optional).** If you tried multiple design variants or parameter settings, summarize the key differences and what improved/degraded performance (a small table or concise figure is fine).
   - **Takeaways.** Briefly interpret the results: what behaviors your navigation terms produced, where they worked well, and where they struggled.

4. **Discussion**
    - What worked and why: link observed behaviors to specific terms/choices.
    - Problems faced & how you solved them: local minima, oscillations, "boundary skimming," tuning instability—be concrete and include short before/after plots if helpful.
    - Trade-offs: safety vs. path length, conservativeness vs. speed, stability vs. responsiveness.
    - If you had more time: 1–2 targeted ideas (e.g., adaptive weights, better pairwise shaping, step-size scheduling).
5. **Conclusion (short paragraph)** Summarize the main design idea, the key evidence it works, and the single most important limitation or next step. Also include what you learned.