TECHNISCHE UNIVERSITÄT WIEN

Stochastic Foundations of Cyber-Physical Systems
182.763 (VU 4,0) Semester: 2025W

Practical Part - Drone Racing RL Project

02-01-2026

Preface

Read all the instructions below carefully before you start working on the assignment, and before you make a submission. All sources of material and resources must be cited.

- Completeness of solution: A complete solution of a task also includes knowledge about the theory behind.
- Grading is on a group basis; you can form groups of up to three students. Upload your solution in TUWEL until 02-02-2026 16:00 as a `.zip` of the project.
- **Indicate the names of your group members in your submitted report!** (Only one person has to submit the report/code.)
- For this assignment, there is no exercise interview. However, if submissions are unclear, students might be invited for exercise interviews.
- A project report should be written and submitted along with your code (in `report.md`).
- **Important hint: The tasks can be solved in parallel.** Task 1 does not have any dependencies. For Task 2 an example rollout is provided in Moodle (`example_rollout.npy`) such that you can test the visualization. For Task 3 you may begin by using the *black-box* environment transition function instead of your own JAX model.

Learning outcomes

The following fundamentals should be understood by the students upon completion of this exercise:

- How to treat a physical system (or black-box simulator) as an MDP and design a learning-compatible interface (`reset`/`step`) with appropriate state, action, and observation representations.
- How to obtain informative identification data from closed-loop interaction (e.g., excitation trajectories) and fit a dynamics model in JAX.
- How to construct observations in task-relevant coordinate frames (gate frame vs. world frame) and why frame choice can strongly affect learning stability and generalization.
- How to design reward functions for continuous-control navigation tasks, including shaping terms (progress), sparse events (gate passes), and regularization (control smoothness, safety constraints).
- How to train and evaluate a PPO agent using a JAX-native RL library (`purejaxrl`) and interpret standard RL diagnostics (return curves, value loss).
- How to improve robustness using practical techniques such as action history features, observation noise, and (optionally) domain randomization.
- How to ensure experimental reproducibility via systematic logging (e.g., `mlflow`) and by comparing both scalar metrics and other diagnostics (e.g., position density).

Deliverables and Submission

Finish the implementation as described below and write a short project report in the `report.md` file. Create a `.zip` folder of the whole project and submit it in TUWEL until 02-02-2026 16:00.

Setup

Install the `uv` Python package manager from: https://github.com/astral-sh/uv

Then install the requirements for this exercise. Note: if you do not have an NVIDIA GPU, do not install *`jax[cuda]`*.

Listing 1 Install required Python packages using uv.

```
# Create and activate a new virtual environment with Python 3.13.2
uv venv --python 3.13.2 .venv
source .venv/bin/activate  # On Linux/macOS
# .venv\Scripts\activate   # On Windows (PowerShell)

# Core dependencies
uv pip install jax optax flax tqdm numpy matplotlib scipy gymnax mlflow rerun-sdk

# Optional (GPU): only if you have CUDA set up
uv pip install "jax[cuda]"
```

1 Handout

This project comes with a pre-compiled environment transition function that acts as a *black-box* simulator of a quadcopter racing drone in *acro mode*. The purpose of the handout is to provide a realistic dynamics source (analogous to a real robot) that you can interact with to collect data and test your agent.

1.1 Coordinate conventions (important)

Throughout this exercise sheet, we use a right-handed world frame with $+z$ up:

$$x: \text{ forward}, \qquad y: \text{ left}, \qquad z: \text{ up}.$$

Yaw is positive for counter-clockwise rotation about $+z$ (turning left when viewed from above). Quaternions are ordered as (q_w, q_x, q_y, q_z).

1.2 Acro mode

In acro mode (also called *rate mode*), the pilot (or agent) directly commands **body angular rates** (roll, pitch, yaw rates). In contrast to self-leveling modes, there is **no automatic stabilization towards level attitude**. If the commanded rates are zero, the drone tends to keep its current attitude rather than returning to level.

In the provided model, the rate commands are mapped linearly to body rates (in degrees/s) and then integrated to update the drone orientation (represented as a quaternion). The thrust command is mapped to a physical thrust force using a battery-voltage-dependent thrust curve. A first-order lag models actuator delays in how RC commands appear as effective rates/throttle.

1.3 Provided files

The handout includes:

- `acro_step.bin`: a serialized binary artifact containing the transition function.
- `acro_step_runtime.py`: a small Python wrapper that loads `acro_step.bin` and exposes a function `step(x,u)` to compute the next state.
- `model.py`: skeleton file for the modeling in Task 1.
- `utils.py`: collection of useful utility functions, such as quaternion helper functions.
- `viz_rerun.py`: skeleton file for the visualization in Task 2.

1.4 How to run the black-box transition

Listing 2 Calling the black-box dynamics transition.

```
from acro_step_runtime import step
x_next = step(x, u)  # x: (21,), u: (4,)
```

1.5 State and action interface

The black-box transition implements a discrete-time dynamical system

$$x_{t+1} = f(x_t, u_t),$$

where the state $x \in \mathbb{R}^{21}$ and action $u \in \mathbb{R}^4$ are structured as:

State x (shape (21,))

$x = [x, y, z,$	position (m)
$v_x, v_y, v_z,$	velocity (m/s)
$a_x, a_y, a_z,$	acceleration (m/s^2)
$q_w, q_x, q_y, q_z,$	orientation quaternion (unit)
$\omega_x, \omega_y, \omega_z,$	angular velocity (rad/s), body frame
$u^{\text{prev}}_{\text{roll}}, u^{\text{prev}}_{\text{pitch}}, u^{\text{prev}}_{\text{yaw}}, u^{\text{prev}}_{\text{thrust}},$	previous action
$V_{\text{bat}}]$	battery voltage (V).

Battery voltage range In this handout, the battery voltage is only valid if it is within the range:

$$V_{\text{bat}} \in [22,\ 24] \text{ Volts.}$$

Be mindful of this range when fitting the thrust model and when implementing safety/termination checks. In the blackbox model the battery discharges at one volt per minute and the blackbox model runs at a timestep of 0.01 seconds (e.g. 100Hz).

Action u (shape (4,))

$$u = [u_{\text{roll}}, u_{\text{pitch}}, u_{\text{yaw}}, u_{\text{thrust}}],$$

where each component is normalized to $[-1, 1]$:

- $u_{\text{roll}}, u_{\text{pitch}}, u_{\text{yaw}}$ represent **rate commands**. They are mapped linearly to angular rates (degrees/s) using fixed maximum rates.
- u_{thrust} represents a **throttle/thrust command**. -1 means no thrust and $+1$ means maximum thrust (as defined by the thrust curve and current battery voltage).

1.6 Important constraint (black-box assumption)

Treat `acro_step.bin` as if it were a real robot: you can query it to obtain (x_t, u_t, x_{t+1}) transitions, but you should assume you do **not** have access to its internal parameters, and you do **not** train your agent by directly backpropagating through the black-box. Instead, you will first identify your own model and implement it in JAX, and then train an RL agent in your own environment.

Task 1: Modeling of Environment Dynamics (20 Points)

Goal

Your goal is to build a JAX dynamics model $\hat{f}_\theta$ that approximates the black-box transition f:

$$x_{t+1} \approx \hat{f}_\theta(x_t, u_t, \Delta t),$$

where θ are **learnable** parameters identified from data collected by interacting with the black-box transition (`acro_step.bin`). The identified model will later be used to train an RL agent in simulation.

Black-box assumption Treat `acro_step.bin` as if it were a real robot: you may query it to obtain transitions (x_t, u_t, x_{t+1}), but you do **not** have access to its internal parameters and you do **not** backpropagate through it. You first identify your own model and then train with that model.

Timing The black-box simulator runs at 100 Hz, i.e. $\Delta t = 0.01$ s. Your learned model must support a general Δt argument, but you may assume 0.01 s for data collection and evaluation unless you explicitly vary it.

What you must implement

Implement a JAX function with the following signature:

Listing 3 Required signature for the learned dynamics.

```
def step(x, u, dt, params):
    """Return x_next given state x, action u, timestep dt, and learned params."""
```

Model parameters (`params`)

Define `params` as a `NamedTuple` that contains both **learnable** parameters (identified from data) and **fixed** parameters.

Learnable parameters

- **First-order command delay time constants** $\tau \in \mathbb{R}^4_{>0}$ (roll, pitch, yaw, thrust). These time constants model how quickly the effective control input follows the commanded input via a first-order lag. Larger τ implies slower response.
- **Thrust curve coefficients** $c \in \mathbb{R}^6$ for a voltage-dependent polynomial thrust model that maps throttle command and battery voltage to thrust force.

Fixed (known) parameters

- **Maximum acro rate vector** (degrees/s):

$$\omega_{\max} = (618.0,\ 618.0,\ 120.0),$$

  corresponding to maximum roll, pitch, yaw rates.
- **Mass** $m = 1.0$ kg.
- **Gravity** $g = 9.80665$ m/s^2 (world frame, $+z$ up so gravity contributes $(0, 0, -g)$).
- **Battery voltage range** $V_{\text{bat}} \in [22, 24]$ V (you may assume this range for identification and evaluation).

Listing 4 Model parameter container (learnable + fixed).

```
class ModelParameters(NamedTuple):  # theta
    # Learnable
    tau: jax.Array              # (4,)  first-order delay time constants (s)
    thrust_coeffs: jax.Array    # (6,)  thrust polynomial coefficients

    # Fixed
    max_rate: jax.Array         # (3,)  deg/s = (618.0, 618.0, 120.0)
    m: float                    # 1.0
    g: float                    # 9.80665
```

Required structure of the learned model

Your model must reproduce the following structure (minimum viable model). You may extend it (e.g. drag) if you clearly document assumptions and fitting.

(1) First-order delay on control inputs (fit τ)

Model actuator/firmware delay as a first-order lag on the commanded action:

$$u_t^{\text{delay}} = u_t^{\text{prev}} + \alpha\,(u_t - u_t^{\text{prev}})\,, \qquad \alpha = 1 - \exp\left(-\Delta t/\tau\right).$$

Here u_t^{prev} is the previous *applied* action stored in the state vector.

(2) Linear acro rate mapping (fixed $\omega_{\max}$)

Assume that delayed rate commands map linearly to body angular rates:

$$\omega^{\text{cmd}} = \text{deg2rad}\left(\omega_{\max} \odot u_{0:3}^{\text{delay}}\right).$$

You must then update the orientation quaternion by integrating the commanded body rates:

$$q_{t+1} = \text{normalize}\left(q_t + \frac{1}{2}\, q_t \otimes [0, \omega^{\text{cmd}}]\, \Delta t\right),$$

where $\otimes$ denotes quaternion multiplication (see `utils.py`).

(3) Polynomial thrust curve (fit c)

Use a battery-voltage-dependent polynomial thrust model:

$$T(u_{\text{th}}, V) = c_0 + c_1 u_{\text{th}} + c_2 u_{\text{th}}^2 + c_3 u_{\text{th}}^3 + c_4 V + c_5 (u_{\text{th}} V).$$

This maps the thrust command $u_{\text{th}} \in [-1, 1]$ and battery voltage V to a physical thrust force T.

(4) Translational dynamics (minimum viable)

At minimum, implement:

$$a_t = R(q_t) \begin{bmatrix} 0 \\ 0 \\ T/m \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix}, \qquad v_{t+1} = v_t + a_t \Delta t, \qquad p_{t+1} = p_t + v_t \Delta t + \tfrac{1}{2} a_t \Delta t^2.$$

You may ignore aerodynamic drag in the minimum solution. If you add drag, state the model and how you fit it.

Data collection and identification

You control what data you record. Use *excitation signals* to make parameters identifiable. Your deliverables must include:

- a script that collects black-box transitions and saves a dataset,
- a fitting script that identifies τ and c.

Practical note on signals The black-box state includes **ideal IMU signals** at the drone center of mass: (a_x, a_y, a_z) and $(\omega_x, \omega_y, \omega_z)$, without noise, drift, or filtering. Use these for system identification.

(A) Fitting delay parameters τ (rate response)

To identify τ for roll/pitch/yaw:

- Excite **one axis at a time** with a sine wave or steps, e.g.

  $$u_{\text{roll}}(t) = A \sin(2\pi f t), \quad u_{\text{pitch}} = u_{\text{yaw}} = 0,$$

  while holding thrust constant (e.g. $u_{\text{thrust}} = -1$ for simplicity).
- For each step, log commanded action u_t and measured body rates $\omega_t = x_t[13:16]$.
- Fit $\tau_{0:3}$ by minimizing the mean squared error between measured ω_t and the model prediction

  $$\hat{\omega}_t = \text{deg2rad}\big(\omega_{\max} \odot u_{t,0:3}^{\text{delay}}\big).$$

(B) Fitting thrust coefficients c (thrust response)

To identify thrust coefficients:

- Keep attitude close to upright and set rate commands to zero.
- Excite u_{thrust} with a sine wave or steps spanning $[-1, 1]$.
- Log battery voltage V_{bat} and world-frame acceleration $a = x[6:9]$.
- **Fit thrust using a near-upright assumption.** If $q \approx (1, 0, 0, 0)$, then world-frame vertical acceleration is approximately

$$a_z \approx \frac{T}{m} - g \quad \Rightarrow \quad T \approx m(a_z + g).$$

  This yields thrust targets T that you can regress onto the polynomial features

$$\phi(u, V) = \big[1,\ u,\ u^2,\ u^3,\ V,\ uV\big].$$

If not upright: If the drone is not near upright, the measured acceleration vector is not aligned with the thrust axis. Then you must (i) rotate the acceleration into a consistent frame using the current attitude (equivalently project onto the thrust axis) and (ii) subtract gravity in the appropriate frame before converting to thrust. Without this correction, attitude-induced mixing biases the fitted thrust curve.

Evaluation and validation

Split your collected transitions into train/validation sets. Report:

- one-step prediction error (e.g., MSE on p, v, q, ω),
- qualitative roll/pitch/yaw response plots (measured vs. predicted),
- thrust fit quality across different battery voltages (22–24 V).

Deliverables for Task 1

Provide:

- your implementation of `step(x,u,dt,params)` in JAX,
- a script to collect data from the black-box transition (`acro_step_runtime.py`),
- a fitting script that learns τ and thrust coefficients (e.g. regression + nonlinear least squares),
- a short description in `report.md` of modeling assumptions, identification procedure, and validation results.

Task 2: Visualization of Rollouts with Rerun (20 Points)

Listing 5 Track coordinates and rotations (world: x forward, y left, z up).

```
# Racing gates: [id, x, y, z, qw, qx, qy, qz]
ENVIRONMENT = jnp.array(
    [
        [0, 12.500000, 2.000000, 1.350000, -0.707107, 0.000000, 0.000000, 0.707107],
        [1,  6.500000, 6.000000, 1.350000, -0.382684, 0.000000, 0.000000, 0.923879],
        ...
    ],
    dtype=jnp.float32,
)
```

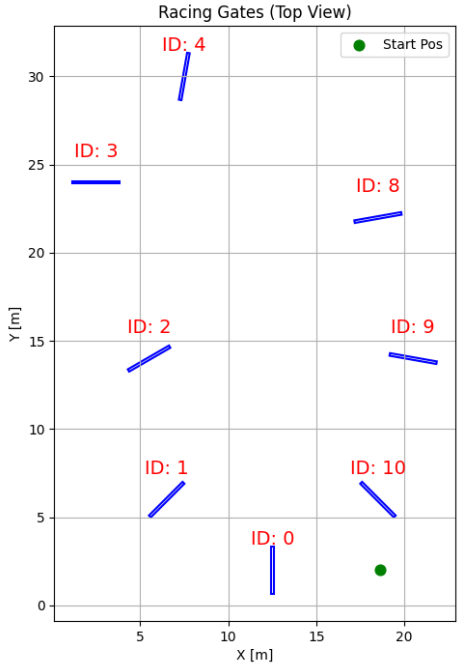


Figure 1: Track Layout

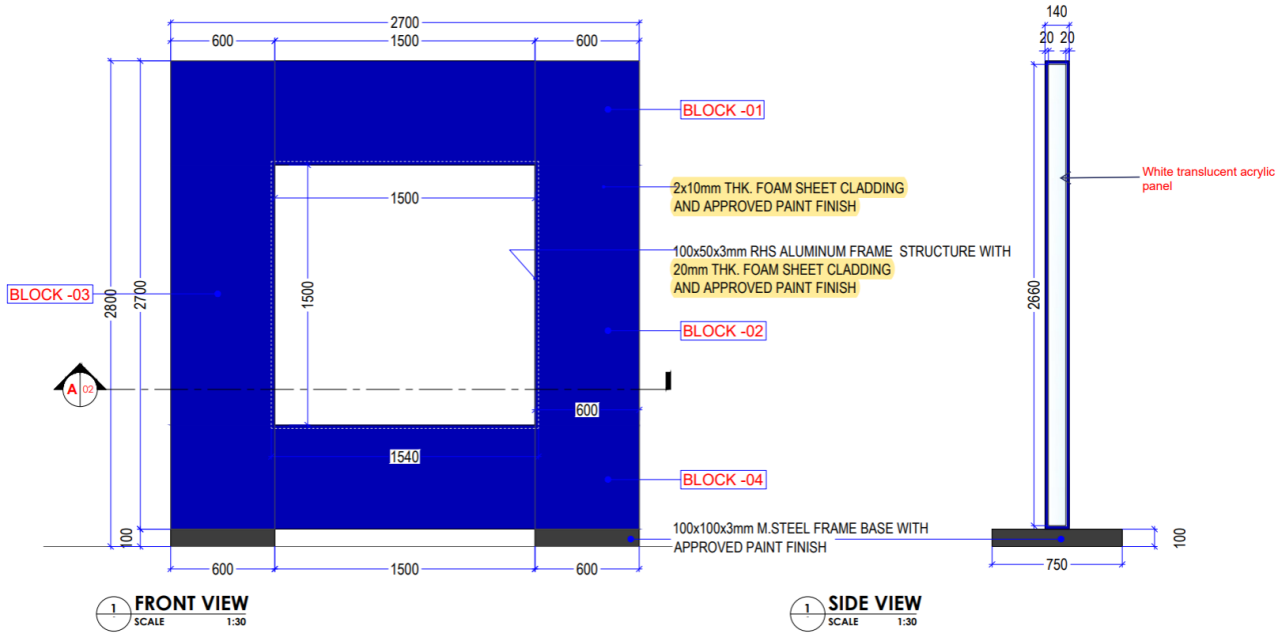


Figure 2: Gate dimensions (you can ignore the foot stands for this exercise)

Goal

In this task you will build a small visualization utility based on `rerun` that allows you to inspect the behavior of your trained agent (and your learned dynamics model). You will implement a script that produces a `.rrd` recording file containing:

- the static race track geometry (gates),
- the drone pose over time (3D visualization),
- time-series plots of relevant scalars (e.g., speed, angular speed, actions).

You can visualize saved `.rrd` files on the website of rerun https://rerun.io/viewer.

Provided components

You are given helper functions that construct the visualization geometry for:

- the racing gates on the track (gate loops / frames),
- the drone body visualization (motors, arms, and a camera ray).

These helpers are extracted from the reference solution and can be treated as correct. Your task is to integrate them into a clean logging pipeline and define a simple interface for visualizing rollouts.

Required interface

Create a module (e.g., `viz_rerun.py`) that exposes the function:

Listing 6 Required visualization function interface.

```
def visualize_state_action_sequence(
    sequence: list[tuple[np.ndarray, np.ndarray]],
    gates: np.ndarray,
    recording_path: str = "rollout.rrd",
    app_id: str = "flytrack_viz",
) -> None:
    """
    Args:
      sequence: list of (x, u) pairs where x: (21,), u: (4,)
      gates: array of gate rows [id, x, y, z, qw, qx, qy, qz], shape (N, 8)
      recording_path: output .rrd file
      app_id: rerun application id
    """
```

Log the following information:

Static scene (logged once)

- World coordinate frame using `rr.ViewCoordinates.RIGHT_HAND_Z_UP`.
- Gate geometry (centers and rectangular frames) using the provided gate helper code and the `gates` argument.

Dynamic rollout (logged for each step)

For each time step $t = 0, \ldots, T-1$:

- Drone pose:
  - position from $p_t = x_t[0:3]$,
  - orientation from $q_t = x_t[9:13]$ (convert to Euler angles, or extend helpers to accept quaternions).
- Drone body visualization (motors, arms, camera ray) using the provided drone helper code.
- A 3D polyline of the trajectory using `rr.LineStrips3D`.

Time-series scalars (recommended)

Log time-series signals to enable quick debugging:

- speed $\|v_t\|_2$ from $v_t = x_t[3:6]$,
- angular speed $\|\omega_t\|_2$ from $\omega_t = x_t[13:16]$,
- action components $u_t[0], \ldots, u_t[3]$ as scalar plots.

Time handling

Use Rerun time to produce synchronized 3D and plot views:

- set a sequence time `step=t` using `rr.set_time("step", sequence=t)`,
- optionally set a physical time axis using the simulator step size (`SIM_DT`) via `rr.set_time("sim_time", duration=t*SIM_DT)`.

Deliverables for Task 2

- `viz_rerun.py` implementing `visualize_state_action_sequence(...)` as specified,
- a short demo script (or a `__main__` block) that loads `example_rollout.npy` (or generates a short random rollout) and writes `rollout.rrd`,
- a brief description in `report.md` on how you built the tool and how you used it.

Hints

- Treat visualization as a debugging tool (sign conventions, quaternion bugs, oscillatory control, gate misses).
- Log static vs. dynamic data correctly: gates and coordinate frames with `static=True`.
- Be explicit about conventions: quaternion order (q_w, q_x, q_y, q_z) and world $+z$ up (gravity $(0, 0, -g)$).

Task 3: Reinforcement Learning with PPO (40 Points)

In this task you will (i) implement a JAX-native racing environment `DroneRaceEnv` based on your learned dynamics model from Task 1, and (ii) train a policy using Proximal Policy Optimization (PPO).

Task 3.1: Implement `DroneRaceEnv` (20 Points)

Create an environment that follows the `gymnax`-style API:

- `reset(rng, params) -> (obs, state)`
- `step(rng, state, action, params) -> (obs, state, reward, done, info)`

The environment must be **fully JAX compatible** (pure functions; no Python side effects) to support `jit`, `vmap`, and `lax.scan`.

Observation design (required)

The observation should be expressed in the **current gate frame**:

- relative position and velocity in the current gate frame,
- orientation relative to the current gate (e.g., Euler angles) and/or angular rates,
- the relative position of the **next gate** (in the current gate frame).

Suggestion: include the previous applied action(s) in the observation to keep the process Markovian when delays or smoothing are present.

Suggested extensions (optional)

- Add observation noise (e.g., on position/velocity/angular rates) to improve robustness.
- Add an action-history buffer (e.g., last 3 applied actions) and expose it in the observation.
- Implement domain randomization (sampled per episode), e.g. randomizing mass, thrust coefficients, drag, or delay constants. Similar to:

  Ferede et al., "One Net to Rule Them All: Domain Randomization in Quadcopter Racing Across Different Platforms", 2025. https://arxiv.org/abs/2504.21586

Reward design (hints)

A typical shaping strategy:

- **Progress reward:** decrease in distance to the current gate between steps.
- **Gate bonus:** a large bonus when the drone passes through a gate (distance below gate radius and/or plane crossing).
- **Penalties:** out-of-bounds, crashes (e.g. too low altitude), missing a gate, timeouts.
- **Regularizers:** control smoothness penalty ($\|u_t - u_{t-1}\|^2$), excess speed penalty, altitude corridor penalty.

Task 3.2: Train a PPO agent using `purejaxrl` (10 Points)

Train a PPO agent using the `purejaxrl` reference implementation:

https://github.com/luchris429/purejaxrl

Requirements

- Use the PPO algorithm as implemented in `purejaxrl`.
- Adapt/wrap your environment so it can be executed in parallel in JAX (i.e., compatible with `vmap` and `lax.scan` as expected by `purejaxrl`).
- Configure training hyperparameters (e.g., number of parallel environments, rollout length, learning rate) and justify your choices briefly in `report.md`.
- Periodically evaluate the policy and export at least one rollout as a sequence of state-action pairs for visualization in Task 2.

Checkpointing Your training pipeline must support continuing training later:

- Save network parameters to disk at the end of training (and optionally periodically).
- Allow resuming from a saved checkpoint via a command-line flag or configuration option (e.g., `--resume path/to/ckpt`).
- Log the final checkpoint as an `mlflow` artifact automatically when training finishes.
- **Hint:** Tools such as "Orbax" might be too complicated for this task and have many unnecessary features if we just want to save our PPO agent. Instead you might just obtain the network parameters and use `flax.serialize.to_bytes` for that task.

Deliverable Provide a runnable training script (e.g., `train_ppo.py`) that trains a PPO agent with `purejaxrl` on your environment, saves a checkpoint, and writes an evaluation rollout for Task 2 visualization.

Task 3.3: Experiment tracking with `mlflow` (10 Points)

Track your experiments using `mlflow`. Your training script should create an `mlflow` run, log the full configuration, and record both scalar learning curves and qualitative diagnostics.

Hint: In PureJaxRL, if `debug=True`, a `jax.debug.callback` can be used to pass rollout summaries to Python. Put `mlflow` logging in that callback. Avoid compute-heavy logging at every step; log expensive artifacts (like density plots) only periodically (e.g. every 100k steps) and/or during evaluation.

Log the following information:

- **Run configuration (parameters):** random seed, environment settings (gate radius, episode length, noise settings), PPO hyperparameters (learning rate, rollout length, number of environments, batch size, number of epochs, clip coefficient, entropy/value coefficients, γ, λ).
- **Training scalars (metrics over training steps):** mean episode return, mean episode length, policy loss, value loss, entropy.
- **Task-specific scalars (metrics):** gates passed per episode, terminations by out-of-bounds vs. time limit, average distance-to-gate.
- **Artifacts (files):** final checkpoint, at least one evaluation rollout (for Task 2), and optionally Rerun `.rrd` recordings.

Position density chart Log a **position density** visualization that shows how frequently the drone visits different regions of the track:

- Collect world-frame positions (x, y) from evaluation rollouts.
- Discretize into **10 cm** $\times$ **10 cm** bins (bin size 0.1 m).
- Compute a 2D histogram (visit counts) and visualize it as a heatmap (e.g. with matplotlib).
- Log the heatmap as an `mlflow` artifact (image).

Task 3.4: Provide an evaluation interface (`eval_agent.py`)

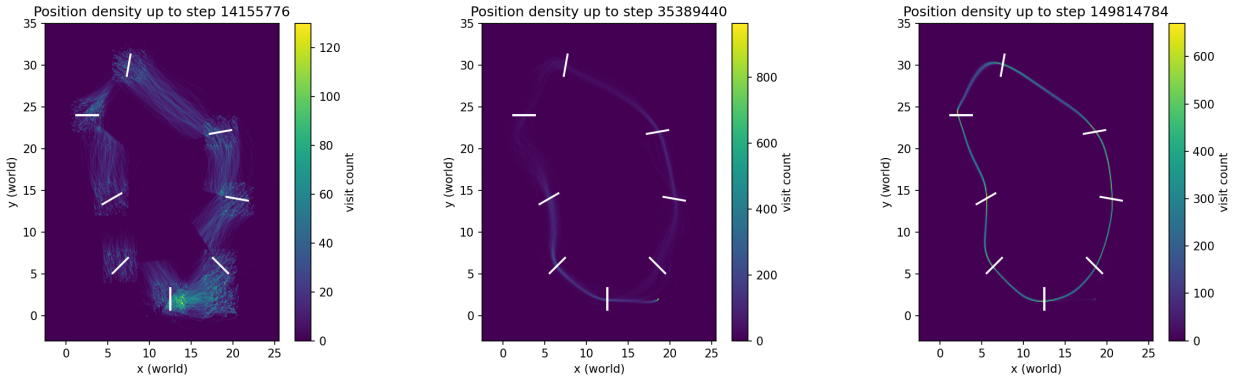
Implement a script `eval_agent.py` that loads a saved policy checkpoint and evaluates it using the **black-box** transition function (`acro_step.bin` via `acro_step_runtime.py`). This is the standardized evaluation interface used for grading task 3, if its not implemented task 3 won't score any points.

Requirements for the eval script:

- Load a saved policy checkpoint from a path argument (e.g. `--checkpoint path`).
- Run the policy for a user-specified number of steps (e.g. `--steps N`) starting from a reset state.
- Use the black-box transition function to advance the state: $x_{t+1} = f(x_t, u_t)$.
- Save an evaluation rollout as a list/array of state-action pairs compatible with Task 2 (e.g. `.npy` file containing a list of (x, u) pairs).

Deliverables for Task 3

- `drone_race_env.py`: your environment implementation (Task 3.1).
- `train_ppo.py`: a runnable PPO training script using `purejaxrl` (Task 3.2), including checkpoint save/resume.
- A policy checkpoint saved to disk and logged to `mlflow`.
- An evaluation rollout saved in a format compatible with Task 2 visualization.
- `eval_agent.py`: standardized evaluation script using the black-box transition (Task 3.4).
- A discussion in `report.md`: reward design, observation design, training curves, and key ablations/lessons learned.



(a) Early training: chaotic exploration. (b) Mid training: gates mostly connected. (c) Final PPO policy: smooth racing line.

Figure 3: Position density chart at different stages of training.