Okay, here is a detailed documentation for the project based on the final version of the code you provided, including the design decisions made along the way.

**Project Documentation: 2D Drone Landing Simulation Environment**

**Version:** (Based on the provided code incorporating metric units, battery, moving platform, randomized start, path/shade rendering, and geometric sensor data)

**Date:** October 26, 2023

**1. Overview**

This project provides a 2D physics-based simulation environment for training Reinforcement Learning (RL) agents to perform a drone landing task. The environment is built using Python and leverages the following libraries:

* **Gym (Gymnasium):** For the standardized RL environment interface (gym.Env).
* **Pymunk:** For 2D rigid body physics simulation (gravity, collisions, forces).
* **Pygame:** For rendering the simulation visually and handling user interactions (like closing the window).
* **NumPy:** For numerical operations, particularly on observations and actions.

The goal for the RL agent is to control the thrust of the drone's two motors to safely land it on a target platform, potentially while dealing with wind, limited battery, and a moving platform.

**2. Features**

* **2D Physics Simulation:** Utilizes Pymunk for realistic interactions, collisions, and gravity.
* **Metric Units:** Operates using meters, kilograms, and seconds for physical parameters, promoting realism and easier translation to real-world scenarios.
* **Configurable Environment:** Allows setting parameters like gravity, wind speed/direction, world dimensions, and maximum steps per episode.
* **Configurable Drone:** Parameters like mass, dimensions, maximum thrust, thrust noise, and battery capacity/consumption rate can be adjusted.
* **Landing Platform:** Features a landing platform with configurable dimensions.
  + **Optional Movement:** The platform can be configured to move horizontally back and forth at a specified speed.
* **Battery Simulation:** Includes a battery level that depletes based on thrust usage, ending the episode if the battery runs out.
* **Wind Simulation:** Applies a constant wind force based on configurable speed, direction, and a simple force coefficient.
* **Gym-Compliant Interface:** Implements the standard gym.Env methods (step, reset, render, close, seed) and defines action\_space and observation\_space.
* **Detailed Observation Space:** Provides the agent with information about:
  + Drone's normalized position, velocity, angle, and angular velocity.
  + Normalized remaining battery level.
  + Cartesian error vector (X, Y distance) to the target platform center.
  + Simulated "Sensor" Data: Normalized relative angle and distance from the drone to the target platform center.
  + (Optional) Normalized horizontal velocity of the platform if moving\_platform is enabled.
* **Continuous Action Space:** The agent outputs continuous values for the thrust of the left and right motors.
* **Reward Shaping:** Provides dense rewards for getting closer to the target and penalizes high velocities, large angles, and excessive angular velocity. Includes significant terminal rewards/penalties for successful landings, crashes, running out of battery, going out of bounds, or exceeding the time limit.
* **Collision Handling:** Detects collisions between the drone and the platform/ground, distinguishing between safe landings (based on velocity and angle thresholds on the platform) and crashes.
* **Visual Rendering (Pygame):**
  + Displays the drone, ground, landing platform, and simulation boundaries.
  + Visualizes wind direction and speed.
  + Shows drone path history.
  + Renders drone "shades" at intervals.
  + Displays key information text (Battery, status, step count, position, velocity, etc.).
  + Optionally displays the randomized initial spawn zone.
* **Randomization:**
  + Random initial drone position within a configurable zone.
  + Random initial drone angle (slight tilt).
  + Random wind direction per episode.
  + Random starting direction for the moving platform.

**3. File Structure**

* **drone\_2d\_env.py (Main Environment File):**
  + Contains the Drone2dEnv class, inheriting from gym.Env.
  + Defines the environment's parameters, state, observation/action spaces.
  + Manages the Pymunk physics space (self.space).
  + Initializes the drone, platform, and ground objects.
  + Implements the core Gym methods (\_\_init\_\_, step, reset, render, close, seed).
  + Handles physics updates, collision detection logic, reward calculation, and state normalization.
  + Contains Pygame rendering logic.
* **Drone.py:**
  + Defines the Drone class.
  + Models the drone's physical structure using Pymunk bodies (frame, left motor, right motor) and joints.
  + Calculates physical properties like moments of inertia based on mass and dimensions.
  + Provides a method (apply\_thrust) to apply forces to the drone model.
* **event\_handler.py:**
  + Contains a simple function (pygame\_events) to handle basic Pygame window events (closing the window via QUIT or ESC key).

**4. Core Concepts & Design Decisions**

1. **Physics Engine (Pymunk):** Chosen for its ease of use within Python, 2D focus matching the project scope, and good integration capabilities with Pygame for rendering. It handles rigid body dynamics, collision detection, and constraints.
2. **Units (Metric):** The decision was made to switch from arbitrary pixel units (often seen in simpler Pygame examples) to metric units (m, kg, s). This makes parameters more intuitive, allows for realistic physical values (gravity = 9.81 m/s²), and makes potential future transition to real-world applications or more complex physics easier. Rendering requires scaling from meters to pixels (pixels\_per\_meter).
3. **Drone Model (Drone.py):**
   * **Multi-Body:** A 3-body model (frame, 2 motors) was used instead of a single body. This allows for more distinct visual representation and potentially more complex physics later (though currently thrust is applied relative to the central frame).
   * **Mass Distribution:** Mass is distributed heuristically (20% frame, 40% each motor). This is a simplification but allows total mass to be a primary parameter.
   * **Joints:** Two PivotJoint constraints per motor connect it rigidly to the frame. This provides stability. error\_bias=0 makes the joints stiff.
   * **Thrust Application:** Thrust is applied via apply\_force\_at\_local\_point on the *central frame body* at points corresponding to the motor arm locations. This simplifies the force application logic compared to applying forces directly to separate motor bodies.
4. **Environment Structure (Drone2dEnv):** Adheres strictly to the gym.Env interface for compatibility with standard RL libraries and algorithms.
5. **State Representation (Observation Space):**
   * **Core State:** Includes the drone's pose (position, angle) and motion (linear/angular velocity), normalized. This is fundamental for control.
   * **Battery:** Included as a critical resource constraint.
   * **Target Information:** The agent needs to know where the target is.
     + **Cartesian Error (err\_x, err\_y):** Initially kept as a simple way to represent the vector to the target. Useful for reward calculation.
     + **"Sensor" Data (Relative Angle & Distance):** Added to simulate how a real drone might perceive the target (polar coordinates relative to the drone). atan2 is used for robust angle calculation. This provides a different perspective than Cartesian error and is arguably more sensor-like. Both Cartesian and relative info are currently provided; one might be removed in future optimization.
   * **Platform Velocity:** Added *only* when the platform is moving, as it's crucial for predicting the target's future location.
   * **Normalization:** Observations are clipped and scaled approximately to [-1, 1] or [0, 1] (for battery/distance). This is standard practice for many RL algorithms to ensure stable learning. Normalization factors (e.g., / 15.0 for velocity) are estimates and might need tuning.
6. **Action Space:** Continuous space [-1, 1] for each motor allows for fine control. The environment scales this to [0, max\_thrust]. Thrust noise is added for a degree of realism/robustness.
7. **Reward Function:** Designed using reward shaping principles:
   * **Dense Proximity Reward:** 2.0 \* np.exp(-0.1 \* dist\_to\_target) strongly encourages minimizing distance. Exponential decay rewards closing large distances more significantly than refining small distances.
   * **Penalties:** Negative rewards for high velocity (encourages smooth movement), large angles (encourages stability), and high angular velocity (discourages spinning). These help guide the agent towards desirable landing behavior.
   * **Terminal Rewards:** Large positive reward for safe, stable landing; smaller positive reward for unstable landing; large negative penalties for failure states (crash, out of bounds, battery empty, time limit). These define the ultimate success/failure.
8. **Collision Handling:** Uses Pymunk collision handlers (begin, separate). The begin handler checks collision types and drone state (velocity, angle) *at the moment of impact* to determine landed\_safely or crashed. Landing requires meeting speed *and* angle thresholds specifically on the platform shape.
9. **Wind Simulation:** A simplified constant global wind force is applied to the drone's main body, scaled by a coefficient and the drone's width (as a proxy for cross-sectional area).
10. **Battery Simulation:** Simple linear depletion based on total thrust applied per time step. Ends the episode immediately upon depletion.
11. **Platform Movement:** Uses a KINEMATIC body, which allows setting velocity directly without being affected by forces (except collisions). Velocity is updated each step (\_update\_platform\_position) to simulate movement and reverse direction at world boundaries.
12. **Rendering (Pygame):** Aims for clarity:
    * Uses \_world\_to\_screen for consistent metric-to-pixel conversion.
    * Draws key elements distinctly.
    * Includes informative text overlays.
    * Path/Shade options provide visual history (aalines for path, blitting rotated/scaled image for shades).
    * Spawn zone visualization aids debugging/understanding.
    * Wind indicator provides immediate feedback on environmental conditions.
13. **Randomization:**
    * **Initial Position:** Randomizes start within a defined square zone (initial\_pos\_random\_range\_m) to increase training robustness and prevent the agent from overfitting to a single start point. Clamping ensures the drone starts within bounds.
    * **Wind Direction:** Randomized per episode for variability.
    * **Platform Start Direction:** Randomized if the platform is moving.

**5. Configuration Parameters (Drone2dEnv.\_\_init\_\_)**

* render\_sim (bool, default=False): If True, initializes Pygame and enables rendering.
* max\_steps (int, default=1000): Maximum number of simulation steps per episode.
* render\_path (bool, default=True): If True and render\_sim is True, draws the drone's trajectory.
* render\_shade (bool, default=True): If True and render\_sim is True, renders drone "shades" based on distance traveled.
* shade\_distance\_m (float, default=2.0): Distance in meters the drone must travel before dropping a new shade.
* moving\_platform (bool, default=False): If True, the landing platform moves horizontally.
* platform\_speed (float, default=1.5): Horizontal speed (m/s) of the platform *if* moving\_platform is True.
* initial\_pos\_random\_range\_m (float, default=5.0): Half-width/height (in meters) of the square zone around the default start point within which the drone's initial position is randomized. Set to 0.0 for a fixed start.
* *(Internal Parameters - Can be modified directly in \_\_init\_\_ for tuning)*:
  + gravity\_mag: Gravitational acceleration.
  + wind\_speed, wind\_force\_coefficient: Wind parameters.
  + lander\_mass, lander\_width, lander\_height: Drone physical properties.
  + initial\_Battery, Battery\_consumption\_rate: Battery parameters.
  + max\_thrust, thrust\_noise\_std\_dev: Motor parameters.
  + world\_width, world\_height, ground\_height: Simulation area dimensions.
  + landing\_pad\_width, landing\_pad\_height: Platform dimensions.
  + max\_safe\_landing\_speed, max\_safe\_landing\_angle: Thresholds for successful landing.
  + frames\_per\_second: Simulation physics update rate.

**6. How to Use**

import gym

# Assuming your package is installed or files are in the right place

# Replace with your actual import path if needed

from drone\_2d\_custom\_gym\_env\_package.drone\_2d\_custom\_gym\_env import Drone2dEnv

import time

# --- Configuration Options ---

RENDER = True

MAX\_EPISODE\_STEPS = 1500

MOVING\_PLATFORM = True

PLATFORM\_SPEED = 2.5

INITIAL\_RANDOM\_RANGE = 10.0 # +/- 10m range

def run\_env():

# Initialize environment with desired options

env = Drone2dEnv(

render\_sim=RENDER,

max\_steps=MAX\_EPISODE\_STEPS,

moving\_platform=MOVING\_PLATFORM,

platform\_speed=PLATFORM\_SPEED,

initial\_pos\_random\_range\_m=INITIAL\_RANDOM\_RANGE

)

print("Action Space:", env.action\_space)

print("Observation Space:", env.observation\_space)

print("Observation Space Sample:", env.observation\_space.sample())

# Run a few episodes

for episode in range(3):

obs = env.reset()

done = False

total\_reward = 0

step\_count = 0

print(f"\n--- Episode {episode + 1} Starting ---")

while not done:

if RENDER:

env.render()

# Add a small delay to make rendering watchable

# time.sleep(0.01)

# --- Replace with your RL agent's action selection ---

# action = agent.predict(obs) # Example placeholder

action = env.action\_space.sample() # Take random actions for now

# ---

# Step the environment

obs, reward, done, info = env.step(action)

total\_reward += reward

step\_count += 1

# Optional: Print step info

# if step\_count % 50 == 0:

# print(f"Step: {step\_count}, Reward: {reward:.3f}, Done: {done}")

# # print(f"Obs: {[f'{x:.2f}' for x in obs]}")

# # print(f"Info: {info}")

if done:

print(f"--- Episode {episode + 1} Finished ---")

print(f"Steps: {step\_count}, Total Reward: {total\_reward:.2f}")

print(f"Final Info: {info}")

if RENDER:

# Keep window open briefly after episode ends

env.render()

time.sleep(1.5)

env.close()

print("\nEnvironment Closed.")

if \_\_name\_\_ == "\_\_main\_\_":

run\_env()

**7. Dependencies**

* gym (or gymnasium)
* numpy
* pygame
* pymunk

**8. Potential Future Enhancements**

* **More Complex Drone Dynamics:** Incorporate aerodynamic drag, lift effects (if applicable in 2D), moment of inertia changes with battery depletion.
* **Advanced Sensors:** Implement raycasting/Lidar simulation (Option 2) or a full pixel-based camera (Option 3). Add sensor noise.
* **Obstacles:** Add static or dynamic obstacles in the environment.
* **Variable Wind:** Implement wind gusts or changes in wind direction/speed during an episode.
* **Uneven Terrain:** Replace the flat ground with a more complex surface.
* **3D Simulation:** Extend the concept to a full 3D environment (would require a different physics engine like PyBullet or MuJoCo and likely a different rendering engine).
* **Different Drone Models:** Simulate different types of drones (e.g., with more rotors, different thrust configurations).
* **Refined Battery Model:** Implement a non-linear battery discharge curve.