Contents

---

1 Dr. Yielding's Questions

While HARL is indeed a subfield of MARL, the distinction between the two can often be unclear or inconsistently used in the literature. In the broadest sense, 'heterogeneous' may refer to any MARL scenario where the agents are not identical. However, using this broad definition risks diluting the practical significance of the term.

To provide a more useful distinction, HARL should be invoked when the heterogeneity of the agents is fundamental, essential, or definitional rather than merely incidental. This means the differences among agents are critical to their roles and interactions within the system, not just minor variations.

Prior to my literature review, I would have considered HARL to apply specifically to cases where agents are distinct from the outset, either in their capabilities (For action-space $\mathcal{A}$, $\mathcal{A}_1 \neq \mathcal{A}_2$) or their observation spaces ($\mathcal{O}_1 \neq \mathcal{O}_2$). This intrinsic heterogeneity is evident in works like [1] (the Irish conference paper mentioned in Dr. Yielding's question) and implicitly reflected in [2] (OpenAI Five), even though they do not explicitly label their methods as HARL.

The most comprehensive source on HARL usage is Zhong et al. [3], whose work has greatly influenced my understanding. They focus on implementing algorithms that encourage the development of heterogeneous policies among agents. Their framework increases the likelihood of individual agents converging on distinct policies, which I term emergent heterogeneity.

In my prospectus, I also mention a suspicion that this approach might not entirely prevent agents from converging on policies that are functionally similar, thus lacking true diversity. However, this assertion remains an ancillary detail as it is not yet substantiated by empirical evidence.

Therefore, I propose distinguishing between intrinsic heterogeneity, where agents are fundamentally different before training, and emergent heterogeneity, where differences arise as a result of the learning process. In the cases where the heterogeneity of the agents falls below a level of functional relevance, and appears to be incidental, I argue that they should not be labeled as HARL. By maintaining these distinctions, we can more accurately categorize and understand the applications and implications of HARL and MARL.

Below, I apply these distinctions to the cases proposed:

1.1 HAA2C and HADDPG are among the numerous algorithms proposed by Zhong et al. [3]. While it seems reasonable that their algorithms could be applied to situations with intrinsic heterogeneity, their implementations and tests are applied to environments with agents that are functionally the same.

I intend to implement (at least a subset of) their algorithms to the extent that time allows. In doing so, I hope to either corroborate or contradict their results by comparing them to similar algorithms under different

conditions. This exploration aims to identify any apparent advantages of these algorithms when applied to the experimental variables proposed for Contribution 1. These experimental variables represent smaller difficulties that we expect to face in Contributions 2 and 3.

1.2 (a) The scenario described in this part of the question is, perhaps unintuitively, more akin to single-agent than multi-agent reinforcement learning. This becomes clearer when considering a single agent acting as an 'overlord,' where the observation space is a combination of observations from all the individual agents. The actions taken by this overlord are combinations of actions chosen for each agent. Essentially, a single policy processes the combined observation and outputs the combined action.

(b) This example is a strong example of MARL, and because the agents utilize copies of a singular policy, this example is free from any of the types of heterogeneity described in the answer for question 1.1.

(c) The types of problems accurately described by the scenario provided in this part of the question appear to be a subset of MARL problems and a superset of HARL problems.

Many MARL algorithms allow the member policies to develop distinctly (e.g., [4]–[6]), but they are not optimized to facilitate the development of distinct policies.

Zhong et al. [3] formulate their series of HARL algorithms with optimizations intended to facilitate the development of distinct policies. One weakness of this formulation is that there is no guarantee that the multiple policies will not converge to a behaviorally indistinct set, similar to the concept of carcinization observed in evolutionary biology.

Thus, the resulting heterogeneity of the agents in these scenarios is not intrinsic but emergent. Whether the algorithm itself is labeled as MARL or HARL is distinguished by intent.

1.3 Referencing Centralized Training Decentralized Execution (CTDE) and Decentralized Training Decentralized Execution (DTDE) as employed by Li et al. in their FA2A paper [7], we see that CTDE is the most common format for Actor-Critic based MARL algorithms [4]–[8].

Li et al. [7] and Wen et al. [9] are the only papers I found that discuss the contrary implementation, DTDE. In both cases, the authors motivate DTDE with practical concerns, particularly the limitations of inter-agent communication in distributed systems. These considerations are important, but the relation to HARL remains the same as described in answer 1.2 (c).

2 Dr. Robbins' Questions

In addition to being included in the appendices of this exam, all code used to run the experiments is available on my github at https://github.com/bhosley/Specialty-Exam. It is written to be run on any ANT-Center VSCode server containers, but should work in a generic virtual environment. The specifications for the virtual machines used for this experiment are enumerated in table 1.

Setting	Value
Template	vscode-server
Image	reg.git.act3-ace.com/ace/hub/vscode-server:v0
Max CPUs	64
Req. CPUs	16
Memory	64 GB
GPU	None

Table 1: Container Settings

I recommended increasing the ratio of memory to CPU allocation for future experiments as the container was substantially closer to maximum utilization of memory than processing for the duration of the experiments run for this exam.

The github readme has a short guide for setting up the virtual environment in the ANT-Center for easy replication. The answers for this section are drawn from the experiment running script, also provided in appendix A of this document.

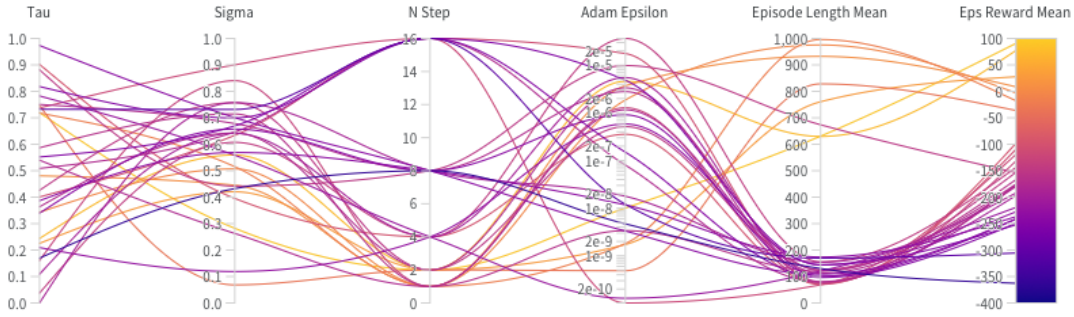


Figure 1: Lunar Lander Baseline Experiment 1 Hyperparameters

2.1 The Deep Q-network (DQN) implementation [10], [11] implemented in the RLlib framework [12] was used for each of the answers in this section. The default results output is readable by tensorboard, but I chose to use the wandb (Weights and Balances) api as well.

To perform this baseline experiment we can call the default script (appendix A). We can accomplish a 5 replication test by setting `--num-env-runners=5`, however, this overrides the default of 10, so we have chosen not to use it and to keep the default. We use the

```
python dqn_exp.py --sweep --num-samples=30 --num-env-runners=60
```

Round 1 - Bad Results:

In the first round of Parameter sweeping, a large number of training runs failed to converge on reasonable results. This was somewhat surprising, and when examining the parameters there was no immediately obvious pattern. Adding the average number of steps per episode to the Comparison (fig. 1) shows consistently shorter episodes for poor performance. Further investigation showed that the observation space assigned to the environment from the RLlib registered lunar lander environment was significantly different from the correct space. Further, manually registering the environment after importing it directly from the Farama maintained packages using Ray's environment registration function resulted in an outdated observation space, less prone to resulting in error, but still problematic.

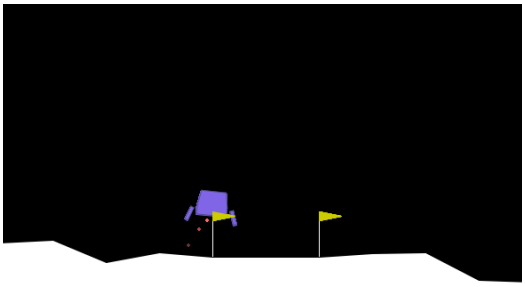
Thus we gain two actionable insights. First, we adjust this experiment to use the custom Lunar Lander environment described in the next section, but set to one agent. Second, we identify an update that can be performed by a pull request to the maintainers of RLlib, a contribution to the open-source software I would like to attempt after the submission of this exam.

Round 2 - Baseline Parameter Sweep:

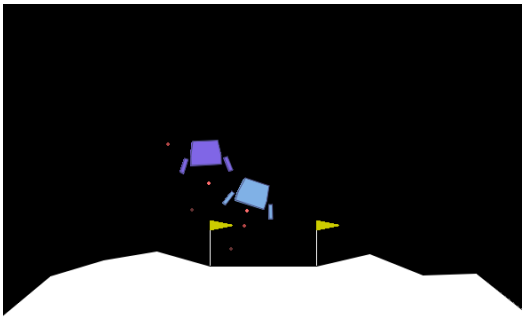
Multi-Agent Lunar Lander

To modify the Lunar Lander environment to support multiple agents it appeared to be possible to simply duplicate all of the sections that referenced the lander in the original code. However, to achieve cleaner code, and behavior more consistent with Farama's PettingZoo environments, I elected to rewrite the environment to be more consistent with object oriented programming principles.

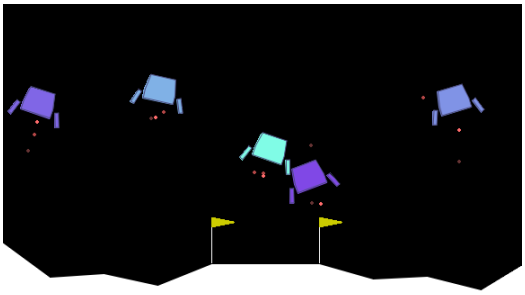
This trivialized the retention of the environmental physics, and made it very easy for the new environment to comply with both the AEC (Agent Environment Cycle; or sequential) and parallel execution methods used in the



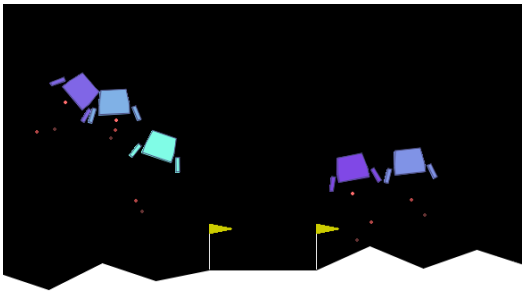
(a) Original



(b) With 2 Landers



(c) With 5 Landers



(d) 5 Landers with Collision

Figure 2: Multi-Agent Lunar Lander

PettingZoo API. Further, this made it much easier to retain the heuristic solution from the original environment. Figure 2 shows several screen shots from the resulting environment when rendered.

While an AEC version of this custom environment was constructed, it was not used for this exam. AEC iterates through the list of agents, allowing the agent to select an action and steps the environment with that action before moving to the next agent, in a manner similar to a board game. Such an implement doesn't necessarily contribute value to the multi agent Lunar Lander sim, and in fact, causes problems if the physics of the environment are not adjusted as the number of landers is increased. Using the heuristic as a measure, the AEC environment becomes unsolvable with 3 or more landers. The effect of their actions must be scaled to overcome the wait time between their turns, otherwise they simple crash, unable to overcome gravity.

The parallel version of the environment was used for this exam, and required only one major assumption, centering around collisions between the landers. Lunar Lander uses the Box2D package to model the moon surface and lander. The most expedient method to address collisions using Box2D was to detect only contact with the lander's body. If any other object touches a lander's body it is considered a crash. As written, the Box2D contact detection does not distinguish between what the secondary object is, if the legs of the lander where included, contact with lunar surface would show as a crash. Thus I chose not to include the legs contact detection for collisions. The effective result is that if two lander's legs touch it is not treated as a crash, however, if one lander's legs touch another's body module it is treated as a crash and thus a failure.

2.2 Formulate a Markov Game: The Markov game that represents the multi-agent version of this environment is an extension of the Markov Decision Process (MDP) that represents the original Lunar Lander environment. The formulation presented here will be used to describe the interactions in all of the questions that follow.

Agent: The agent is represented as a member of set of agents $n \in N$.

State Space: Let $\mathcal{S}$ be the state space. Then, $\mathcal{S} \equiv s^N$, where

$$s = \begin{cases} x \in [-2.5, 2.5] & \text{Position of } n \text{ in } x \\ y \in [-2.5, 2.5] & \text{Position of } n \text{ in } y \\ \vec{x} \in [-10, 10] & \text{Velocity of } n \text{ in } x \\ \vec{y} \in [-10, 10] & \text{Velocity of } n \text{ in } y \\ \omega \in [-2\pi, 2\pi] & \text{Angle of } n \\ \vec{\omega} \in [-10, 10] & \text{Angular Velocity of } n \\ \mathbb{I}(\text{leg 1}) \in \{0, 1\} & \text{Leg on ground} \\ \mathbb{I}(\text{leg 2}) \in \{0, 1\} & \text{Leg on ground} \end{cases}$$

Action Space: Let $\mathcal{A}$ be the action space. Then, $\mathcal{A} \equiv a^N$, where

$$a \in \begin{cases} 0 & \text{No-Op} \\ 1 & \text{Fire Left Engine} \\ 2 & \text{Fire Main Engine} \\ 3 & \text{Fire Right Engine} \end{cases}$$

Transition Probability: The transition probability for each agent n remains the same as the original environment,

$$P_n(s_{n,t+1}|s_{n,t}, a_{n,t}) \cong \begin{cases} \text{Dispersion} \sim U(-1, 1) \\ \text{Wind(Linear)} = \tanh(\sin(2kx) + \sin(\pi kx)) \\ \text{Wind(Rotate)} = \tanh(\sin(2kx) + \sin(\pi kx)) \end{cases}$$

Thus the transition probability for the game as a whole is

$$P(s_{t+1}|s_t, a_t) = \prod_{n \in N} P_n$$

Reward: The reward structure is similarly retained from the original Lunar Lander environment. Specifically, the reward at time t is defined as $r(t) = \sum_{n \in N} r_n(t)$ where

$$\begin{aligned} r_n(t) = & \\ & \pm 100 && \text{End-state, crash or land} \\ & + 10(s_{t,6} + s_{t,7}) && \text{leg(s) on ground} \\ & - a_n(t) \cdot [0, 0.03, 0.3, 0.03] && \text{thruster cost} \\ & - 100\sqrt{s_n(t)_0^2 + s_n(t)_1^2} && \text{Distance} \\ & - 100\sqrt{s_n(t)_2^2 + s_n(t)_3^2} && \text{Velocity} \\ & - 100|\omega_n(t)| && \text{Tilt} \end{aligned}$$

2.3 DQN Single-Agent Controller: This problem was approached using a wrapper class around the custom multi-agent environment to translate the between the multi-agent environment and single-agent policy. The proxy single-agent state/observation was formed by simply concatenating the state vectors of each agent.

The modified action space for this problem is $\mathcal{B} \equiv \mathcal{A}^N$. To relate the two action spaces for this problem I use the function,

$$b = \sum a_n \times \dim(a)^n.$$

Then to translate the modified action into the agent-action vector necessary for the multi-agent environment I use the following function,

$$\mathbf{a} = \{b/\dim(a)^n \mod \dim(a)\}_{n \in N}$$

which relies on $\dim(\mathcal{A}_n) = \dim(\mathcal{A}_m) \forall\ n, m \in N$, an assumption that I note as it is true for this problem, limits some generality. The policy can thus be represented as $\pi(b \big| [s_n]_{n \in N})$.

Finally, the experiment can be replicated using the command:

```
python dqn_exp.py --SA --sweep --num-samples=30 --num-env-runners=30
```

2.4 NoPS - No Parameter Sharing: This experiment uses a distinct policy for each agent, $\pi_n(a_n|s_n)$, and can be replicated using:

```
python src/dqn_exp.py --NoPS --sweep --num-samples=30 --num-env-runners=30
```

2.5 FuPS - Full Parameter Sharing: This experiment uses a shared policy for each agent, $\pi(a_n|s_n)$, and can be replicated using:

```
python src/dqn_exp.py --FuPS --sweep --num-samples=30 --num-env-runners=30
```

2.6 Results Comparisons:

3 Dr. Cox's Questions

A Experiment Running Script

```
"""Script for Dr. Robbins Questions
Establishing a baseline implementation of Single Agent Lunar Lander
with a DQN.

See: https://gymnasium.farama.org/environments/box2d/lunar_lander/
for more details on the environment.

How to run this script
----------------------
`python [script_name].py`

For debugging, use the following additional command line options
`--no-tune --num-env-runners=0`

For logging to your WandB account, use:
`--wandb-key=[your WandB API key] --wandb-project=[some project name]
--wandb-run-name=[optional: WandB run name (within the defined project)]`




python dqn_exp.py \
    --num-env-runners=5 \
    --stop-reward=200 \
    --wandb-key=913528a8e92bf601b6eb055a459bcc89130c7f5f \
    --wandb-project=lunar_lander_test

tensorboard --logdir logs/fit
"""
# --evaluation-duration=5 \ # Default uses 10
# --checkpoint-at-end
# --num-samples=30
# result_dict['env_runners']['episode_return_mean']


import numpy as np
from gymnasium import Wrapper, spaces
from argparse import ArgumentParser

from ray import tune
from ray.rllib.core.rl_module.rl_module import SingleAgentRLModuleSpec
from ray.rllib.core.rl_module.marl_module import MultiAgentRLModuleSpec
from ray.rllib.env.wrappers.pettingzoo_env import ParallelPettingZooEnv
from ray.rllib.utils.test_utils import (add_rllib_example_script_args,
                                        run_rllib_example_script_experiment)
from ray.tune.registry import get_trainable_cls, register_env

import multi_lander


parser = add_rllib_example_script_args(
    ArgumentParser(conflict_handler='resolve'), # Resolve for num_agents
    default_reward=0.0,
    default_iters=100, #100, #200
    default_timesteps=100000, #10000, #100000
)
```

```python
parser.add_argument(
    "--control",
    type=str,
    choices=["Baseline", "SA", "NoPS", "FuPS"],
    default="Baseline",
    help="The controller method."
    "`Baseline`: Original Lunar Lander with single agent and lander"
    "`SA`: A single agent controls all of the landers."
    "`NoPS`: No parameter sharing between the agents."
    "`FuPS`: Full parameter sharing between the agents.",
)
parser.add_argument(
    "--SA", action="store_true",
    help="`SA`: Creates a single agent for controlling all of the agents.",
)
parser.add_argument(
    "--NoPS", action="store_true",
    help="`NoPS`: No parameter sharing between the agents.",
)
parser.add_argument(
    "--FuPS", action="store_true",
    help="`FuPS`: Full parameter sharing between the agents.",
)
parser.add_argument(
    "--num-agents", type=int, default=2,
    help="The number of agents",
)
parser.add_argument(
    "--sweep", action="store_true",
    help="Perform a parameter sweep instead of using tune",
)


class CustomWrapper(ParallelPettingZooEnv):
    """Wraps a Parallel Petting Zoo Environment for RLlib

    This wrapper is necessary to interface with rllib's workers.
    There is a problem caused when the one agent terminates before
    the other, which seems to pass inconsistent length episode
    trajectories to the rollout worker(s).

    It also adds the necessary `__all__` key to the done dictionaries.
    """
    def step(self, action_dict) -> tuple:
        obs, rew, terminateds, truncateds, info = self.par_env.step(action_dict)

        active = [a_id for a_id,term in terminateds.items() if term==False]

        obs_s = {a_id: obs.get(a_id) for a_id in active}
        rew_s = {a_id: rew.get(a_id) for a_id in active}
        terminateds = {a_id: terminateds.get(a_id) for a_id in active}
        truncateds = {a_id: truncateds.get(a_id) for a_id in active}

        terminateds["__all__"] = all(terminateds.values())
        truncateds["__all__"] = all(truncateds.values())
        return obs_s, rew_s, terminateds, truncateds, info

# Registry is necessary for functional passing later.
register_env("ma-lander", lambda _: CustomWrapper(multi_lander.Parallel_Env()))
```

```python

class SingleAgentWrapper(Wrapper):
    """Wraps a parallel multi-agent environment for single-agent control.

    *It is currently limited to agents with discrete action spaces and
    box observation spaces with identical lengths.

    This will wrap environments with an arbitrary number of agents,
    however, the action space scales exponentially by the number of
    agents and is likely to be the limiting factor for performance.
    """
    def __init__(self,*args):
        super().__init__(*args)
        # Concat all action spaces into a single action space
        self.action_space = spaces.Discrete(np.prod(
                            [list(self.env.action_spaces.values())[i].n
                            for i in range(len(self.env.observation_spaces))]))
        self._act_n = list(self.env.action_spaces.values())[0].n
        # Concat all obs spaces into a single obs space
        low,high = [],[]
        for i in range(len(self.env.observation_spaces)):
            low.extend(list(self.env.observation_spaces.values())[i].low)
            high.extend(list(self.env.observation_spaces.values())[i].high)
        self.observation_space = spaces.Box(np.array(low), np.array(high))

    def observe(self) -> tuple:
        # Concat all obs values into a single obs tuple
        #return sum(self.env.observe().values(),[]) # By monoid
        return np.array(list(self.env.observe().values())).flatten()

    def observation(self) -> tuple:
        return self.observe()

    def reset(self,*, seed:int | None=None, options:dict | None=None) -> tuple:
        self.env.reset(seed=seed, options=options)
        return self.observe(), {}

    def step(self, action:int) -> tuple:
        action_list = {a: divmod(int(action/(self._act_n**i)),self._act_n)[1]
                       for i,a in enumerate(self.agents)}
        obss, rews, terminations, _, _ = self.env.step(action_list)
        obs = self.observe()
        reward = sum(rews.values())
        term = all(terminations.values()) or self.env.game_over
        return obs, reward, term, False, {}

# Registry is necessary for functional passing later.
register_env("sa-lander",
    lambda _: SingleAgentWrapper(multi_lander.Parallel_Env()))


if __name__ == "__main__":
    args = parser.parse_args()

    # Shared config settings for all experiments
    base_config = (
        get_trainable_cls("DQN")
        .get_default_config()
        .framework('torch')
    )
```

```
    # Set config for experiment if using Single Agent Control
    if args.control == 'NoPS' or args.SA:
        config = (
            base_config
            .environment(
                "sa-lander",
                env_config={"num_landers": args.num_agents}
            )
        )
    # Set config for experiment if using No-Parameter Sharing
    elif args.control == 'NoPS' or args.NoPS:
        policies = {"lander_" + str(i) for i in range(args.num_agents)}
        config = (
            base_config
            .multi_agent(
                policies=policies, # 1:1 map from AgentID to ModuleID.
                policy_mapping_fn=(lambda aid, *args, **kwargs: aid),
            )
            .rl_module(
                rl_module_spec=MultiAgentRLModuleSpec(
                   module_specs={p:SingleAgentRLModuleSpec() for p in policies},
                ),
            )
            .environment(
                "ma-lander",
                env_config={"num_landers": args.num_agents}
            )
        )
    # Set config for experiment if using Full-Parameter Sharing
    elif args.control == 'FuPS' or args.FuPS:
        config = (
            base_config
            .multi_agent(
                policies={"p0"}, # All agents map to the same policy.
                policy_mapping_fn=(lambda aid, *args, **kwargs: "p0"),
            )
            .rl_module(
                rl_module_spec=MultiAgentRLModuleSpec(
                    module_specs={"p0": SingleAgentRLModuleSpec()},
                ),
            )
            .environment(
                "ma-lander",
                env_config={"num_landers": args.num_agents}
            )
        )
    # Default, original lunar lander with one agent and lander
    else:
        config = (
        base_config
        .environment(
            #env="LunarLander-v2"
            "ma-lander",
            env_config={"num_landers": 1}
        )
    )

    # Parameter sweep settings
    if args.sweep:
```

```
        param_space = {
            "adam_epsilon": tune.loguniform(1e-4, 1e-10), # 1e-8
            "sigma0": tune.randn(0.5, 0.2), # 0.5
            "n_step": tune.choice([2**i for i in range(5)]), # 1
            # Update the target by \tau * policy + (1-\tau) * target_policy.
            "tau": tune.uniform(0.0,1.0), # 1.0,
            #epsilon = [(0, 1.0), (10000, 0.05)]    [(step,epsilon),...]
            # -> 1.0 at beginning, decreases to 0.05 over 10k steps
        }
        config = config.training(**param_space)
        # Use Param sweep, not tune
        #args.no_tune = True

    # Call experiment runner
    run_rllib_example_script_experiment(config, args)
```

B Multi-Agent Lunar Lander

```python
import math
from typing import TYPE_CHECKING, Optional
import numpy as np

import gymnasium as gym
from gymnasium import spaces, logger
from gymnasium.error import DependencyNotInstalled
from gymnasium.utils import EzPickle

from pettingzoo import AECEnv
from pettingzoo.utils import agent_selector

try:
    import Box2D
    from Box2D.b2 import (
        circleShape,
        contactListener,
        edgeShape,
        fixtureDef,
        polygonShape,
        revoluteJointDef,
    )
except ImportError as e:
    raise DependencyNotInstalled(
        'Box2D is not installed, you can install it by run `pip install swig` '
        + 'followed by `pip install "gymnasium[box2d]"`'
    ) from e

if TYPE_CHECKING:
    import pygame

FPS = 50
SCALE = 30.0  # affects how fast the game is, forces should be adjusted as well
MAIN_ENGINE_POWER = 13.0
SIDE_ENGINE_POWER = 0.6

INITIAL_RANDOM = 1000.0  # Set 1500 to make game harder

LANDER_POLY = [(-14, +17), (-17,0), (-17, -10), (+17, -10), (+17,0), (+14, +17)]
LEG_AWAY = 20
LEG_DOWN = 18
LEG_W, LEG_H = 2, 8
LEG_SPRING_TORQUE = 40

SIDE_ENGINE_HEIGHT = 14
SIDE_ENGINE_AWAY = 12
MAIN_ENGINE_Y_LOCATION = 4 # The Y loc of the main engine on the lander body.

VIEWPORT_W = 600
VIEWPORT_H = 400


class ContactDetector(contactListener):
    def __init__(self, env):
        contactListener.__init__(self)
        self.env = env

    def BeginContact(self, contact):
```

```
        for lander in self.env.landers:
            if (
                lander == contact.fixtureA.body
                or lander == contact.fixtureB.body
            ):
                self.env.game_over = True
        for leg in self.env.legs:
            if leg in [contact.fixtureA.body, contact.fixtureB.body]:
                leg.ground_contact = True

    def EndContact(self, contact):
        for leg in self.env.legs:
            if leg in [contact.fixtureA.body, contact.fixtureB.body]:
                leg.ground_contact = False


class SequentialEnv(AECEnv, EzPickle):
    r"""
    This is a rewrite of the Farama foundation Lunar Lander environment from:
        https://gymnasium.farama.org/environments/box2d/lunar_lander/
    Adapted for a AEC type petting zoo environment.
    The action space and physics remain the same.
    """
    metadata = {
        "render_modes": ["human", "rgb_array"],
        "name": "multi_lander_v0",
        "is_parallelizable": True,
        "render_fps": FPS,
    }

    def __init__(
            self,
            render_mode: Optional[str] = None,
            continuous: bool = False,
            gravity: float = -10.0,
            enable_wind: bool = False,
            wind_power: float = 15.0,
            turbulence_power: float = 1.5,
            num_landers: int = 2,
            *args, **kwargs
    ):
        EzPickle.__init__(
            self,
            render_mode,
            continuous,
            gravity,
            enable_wind,
            wind_power,
            turbulence_power,
            *args, **kwargs
        )
        AECEnv.__init__(self)
        self.render_mode = render_mode
        self.np_random = np.random

        # Value Checkers
        assert (-12.0 < gravity and gravity < 0.0
            ), f"gravity (current value: {gravity}) must be between -12 and 0"
        self.gravity = gravity

```

```
        if 0.0 > wind_power or wind_power > 20.0:
            logger.warn(
                "wind_power value is recommended to be between 0.0 and 20.0, " +
                f"(current value: {wind_power})"
            )
        self.wind_power = wind_power

        if 0.0 > turbulence_power or turbulence_power > 2.0:
            logger.warn(
                "turbulence_power value is recommended to be between 0.0 and " +
                f"2.0, (current value: {turbulence_power})"
            )
        # These are bounds for position realistically the environment
        # should have ended long before we reach more than 50% outside
        low = np.array(
            [
                -2.5, -2.5,             # x,y coordinates
                -10.0, -10.0,           # x,y velocity bounds is 5x rated speed
                -2 * math.pi, -10.0,    # Angle, Angular Velocity
                -0.0, -0.0,             # L,R Leg not on ground
            ]).astype(np.float32)
        high = np.array(
            [
                2.5, 5.0, # 2.5,        # x,y coordinates
                10.0, 10.0,             # x,y velocity bounds is 5x rated speed
                2 * math.pi, 10.0,      # Angle, Angular Velocity
                1.0, 1.0,               # L,R Leg on ground
            ]).astype(np.float32)

        # Environmental Variables
        self.turbulence_power = turbulence_power
        self.enable_wind = enable_wind
        self.screen: pygame.Surface = None
        self.clock = None
        self.isopen = True
        self.world = Box2D.b2World(gravity=(0, gravity))
        self.moon = None
        self.particles = []
        self.landers = []
        self.legs = []
        self.prev_reward = None
        self.continuous = continuous

        # Agents
        self.possible_agents = ["lander_" + str(i) for i in range(num_landers)]
        self.agents = self.possible_agents[:]
        self.agent_name_mapping = dict(zip(self.agents,
                                           list(range(self.num_agents))))
        self._agent_selector = agent_selector(self.agents)

        # Spaces
        self.observation_spaces = dict(zip(self.agents,
                                       [spaces.Box(low, high)]*self.num_agents))
        if self.continuous:
            # Action is two floats [main engine, left-right engines].
            # Main engine: -1..0 off, 0..+1 throttle from 50% to 100% power.
            #                 Engine can't work with less than 50% power.
            # Left-right:  -1.0..-0.5 fire left engine, +0.5..+1.0 fire right
            #                 engine, -0.5..0.5 off
            self._action_space = spaces.Box(-1, +1, (2,), dtype=np.float32)
```

```
        else:
            # No-op, fire left engine, main engine, right engine
            self._action_space = spaces.Discrete(4)
        self.action_spaces = dict(zip(self.agents,
                                      [self._action_space]*self.num_agents))

    def observation_space(self, agent):
        return self.observation_spaces[agent]

    def action_space(self, agent):
        return self.action_spaces[agent]

    def convert_to_dict(self, list_of_list):
        return dict(zip(self.agents, list_of_list))

    def close(self):
        if self.screen is not None:
            import pygame
            pygame.display.quit()
            pygame.quit()
            pygame.QUIT
            self.isopen = False

    def create_landers(self):
        landers = []
        for n in range(self.num_agents):
            lander: Box2D.b2Body = self.world.CreateDynamicBody(
                position=((n+1)/(self.num_agents+1) * VIEWPORT_W / SCALE,
                          VIEWPORT_H / SCALE),
                angle=0.0,
                fixtures=fixtureDef(
                    shape=polygonShape(
                        vertices=[(x/SCALE, y/SCALE) for x, y in LANDER_POLY]),
                    density=5.0,
                    friction=0.1,
                    categoryBits=0x0010,
                    #maskBits=0x001,  # uncomment for : collide only with ground
                    restitution=0.0,
                ),  # 0.99 bouncy
            )
            lander.color1 = ((128), (102+75*(n))%255, (230))
            lander.color2 = ((77), (77), (128))
            landers.append(lander)
        self.landers = landers

        legs = []
        for n,lander in enumerate(self.landers):
            for i in [-1, +1]:
                leg = self.world.CreateDynamicBody(
                    position=(lander.position[0] - i * LEG_AWAY / SCALE,
                              lander.position[1]),
                    angle=(i * 0.05),
                    fixtures=fixtureDef(
                        shape=polygonShape(box=(LEG_W / SCALE, LEG_H / SCALE)),
                        density=1.0,
                        restitution=0.0,
                        categoryBits=0x0020,
                        #maskBits=0x001,
                    ),
                )
```

```
                leg.ground_contact = False
                leg.color1 = ((128), (102+75*(n))%255, (230))
                leg.color2 = ((77), (77), (128))
                rjd = revoluteJointDef(
                    bodyA=lander,
                    bodyB=leg,
                    localAnchorA=(0, 0),
                    localAnchorB=(i * LEG_AWAY / SCALE, LEG_DOWN / SCALE),
                    enableMotor=True,
                    enableLimit=True,
                    maxMotorTorque=LEG_SPRING_TORQUE,
                    motorSpeed=+0.3 * i,  # low enough not to jump into the sky
                )
                if i == -1:
                    rjd.lowerAngle = (
                        +0.9 - 0.5
                    )  # Valid angle of travel for the legs
                    rjd.upperAngle = +0.9
                else:
                    rjd.lowerAngle = -0.9
                    rjd.upperAngle = -0.9 + 0.5
                leg.joint = self.world.CreateJoint(rjd)
                legs.append(leg)
        self.legs = legs

    def _create_particle(self, mass, x, y, ttl):
        p = self.world.CreateDynamicBody(
            position=(x, y),
            angle=0.0,
            fixtures=fixtureDef(
                shape=circleShape(radius=2 / SCALE, pos=(0, 0)),
                density=mass,
                friction=0.1,
                categoryBits=0x0100,
                maskBits=0x001,  # collide only with ground
                restitution=0.3,
            ),
        )
        p.ttl = ttl
        self.particles.append(p)
        self._clean_particles(False)
        return p

    def _clean_particles(self, all_particle):
        while self.particles and (all_particle or self.particles[0].ttl < 0):
            self.world.DestroyBody(self.particles.pop(0))


    def render(self):
        if self.render_mode is None:
            assert self.spec is not None
            gym.logger.warn(
                "You are calling render method without specifying render mode."
                "You can specify the render_mode at initialization, "
                f'e.g. gym.make("{self.spec.id}", render_mode="rgb_array")')
            return

        try:
            import pygame
            from pygame import gfxdraw
```

```
        except ImportError as e:
            raise DependencyNotInstalled(
                'pygame is not installed, run `pip install "gymnasium[box2d]"`'
            ) from e

        if self.screen is None and self.render_mode == "human":
            pygame.init()
            pygame.display.init()
            self.screen = pygame.display.set_mode((VIEWPORT_W, VIEWPORT_H))
        if self.clock is None:
            self.clock = pygame.time.Clock()

        self.surf = pygame.Surface((VIEWPORT_W, VIEWPORT_H))
        pygame.transform.scale(self.surf, (SCALE, SCALE))
        pygame.draw.rect(self.surf, (255, 255, 255), self.surf.get_rect())

        for obj in self.particles:
            obj.ttl -= 0.15
            obj.color1 = (
                int(max(0.2, 0.15 + obj.ttl) * 255),
                int(max(0.2, 0.5 * obj.ttl) * 255),
                int(max(0.2, 0.5 * obj.ttl) * 255),
            )
            obj.color2 = (
                int(max(0.2, 0.15 + obj.ttl) * 255),
                int(max(0.2, 0.5 * obj.ttl) * 255),
                int(max(0.2, 0.5 * obj.ttl) * 255),
            )

        self._clean_particles(False)

        for p in self.sky_polys:
            scaled_poly = []
            for coord in p:
                scaled_poly.append((coord[0] * SCALE, coord[1] * SCALE))
            pygame.draw.polygon(self.surf, (0, 0, 0), scaled_poly)
            gfxdraw.aapolygon(self.surf, scaled_poly, (0, 0, 0))

        for obj in self.particles + self.drawlist:
            for f in obj.fixtures:
                trans = f.body.transform
                if type(f.shape) is circleShape:
                    pygame.draw.circle(
                        self.surf,
                        color=obj.color1,
                        center=trans * f.shape.pos * SCALE,
                        radius=f.shape.radius * SCALE,
                    )
                    pygame.draw.circle(
                        self.surf,
                        color=obj.color2,
                        center=trans * f.shape.pos * SCALE,
                        radius=f.shape.radius * SCALE,
                    )
                else:
                    path = [trans * v * SCALE for v in f.shape.vertices]
                    pygame.draw.polygon(self.surf,color=obj.color1,points=path)
                    gfxdraw.aapolygon(self.surf, path, obj.color1)
                    pygame.draw.aalines(
                        self.surf, color=obj.color2, points=path, closed=True)
```

```
359
360                 for x in [self.helipad_x1, self.helipad_x2]:
361                     x = x * SCALE
362                     flagy1 = self.helipad_y * SCALE
363                     flagy2 = flagy1 + 50
364                     pygame.draw.line(
365                         self.surf,
366                         color=(255, 255, 255),
367                         start_pos=(x, flagy1),
368                         end_pos=(x, flagy2),
369                         width=1,
370                     )
371                     pygame.draw.polygon(
372                         self.surf,
373                         color=(204, 204, 0),
374                         points=[
375                             (x, flagy2),
376                             (x, flagy2 - 10),
377                             (x + 25, flagy2 - 5),
378                         ],
379                     )
380                     gfxdraw.aapolygon(
381                         self.surf,
382                         [(x, flagy2), (x, flagy2 - 10), (x + 25, flagy2 - 5)],
383                         (204, 204, 0),
384                     )
385
386         self.surf = pygame.transform.flip(self.surf, False, True)
387
388         if self.render_mode == "human":
389             assert self.screen is not None
390             self.screen.blit(self.surf, (0, 0))
391             pygame.event.pump()
392             self.clock.tick(self.metadata["render_fps"])
393             pygame.display.flip()
394         elif self.render_mode == "rgb_array":
395             return np.transpose(
396                 np.array(pygame.surfarray.pixels3d(self.surf)), axes=(1, 0, 2)
397             )
398     # End Render()
399
400     def _destroy(self):
401         if not self.moon:
402             return
403         self.world.contactListener = None
404         self._clean_particles(True)
405         self.world.DestroyBody(self.moon)
406         self.moon = None
407         for lander in self.landers:
408             self.world.DestroyBody(lander)
409             lander = None
410         for i in range(len(self.legs)):
411             self.world.DestroyBody(self.legs[i])
412
413
414     def reset(self,*,seed: Optional[int] = None,options: Optional[dict] = None):
415         self._destroy()
416         # Issue: https://github.com/Farama-Foundation/Gymnasium/issues/728
417         # self._destroy() is not enough to clean(reset), workaround is
418         # to create a totally new world for self.reset()
```

```
        self.world = Box2D.b2World(gravity=(0, self.gravity))
        self.world.contactListener_keepref = ContactDetector(self)
        self.world.contactListener = self.world.contactListener_keepref
        self.game_over = False
        self.prev_shaping = [None]*self.num_agents

        W = VIEWPORT_W / SCALE
        H = VIEWPORT_H / SCALE

        # Create Terrain
        CHUNKS = 11
        height = self.np_random.uniform(0, H / 2, size=(CHUNKS + 1,))
        chunk_x = [W / (CHUNKS - 1) * i for i in range(CHUNKS)]
        self.helipad_x1 = chunk_x[CHUNKS // 2 - 1]
        self.helipad_x2 = chunk_x[CHUNKS // 2 + 1]
        self.helipad_y = H / 4
        height[CHUNKS // 2 - 2] = self.helipad_y
        height[CHUNKS // 2 - 1] = self.helipad_y
        height[CHUNKS // 2 + 0] = self.helipad_y
        height[CHUNKS // 2 + 1] = self.helipad_y
        height[CHUNKS // 2 + 2] = self.helipad_y
        smooth_y = [
            0.33 * (height[i - 1] + height[i + 0] + height[i + 1])
            for i in range(CHUNKS)
        ]

        self.moon = self.world.CreateStaticBody(
            shapes=edgeShape(vertices=[(0, 0), (W, 0)])
        )
        self.sky_polys = []
        for i in range(CHUNKS - 1):
            p1 = (chunk_x[i], smooth_y[i])
            p2 = (chunk_x[i + 1], smooth_y[i + 1])
            self.moon.CreateEdgeFixture(vertices=[p1,p2],density=0,friction=0.1)
            self.sky_polys.append([p1, p2, (p2[0], H), (p1[0], H)])

        self.moon.color1 = (0.0, 0.0, 0.0)
        self.moon.color2 = (0.0, 0.0, 0.0)

        if self.enable_wind:  # Initialize wind pattern based on index
            self.wind_idx = self.np_random.integers(-9999, 9999)
            self.torque_idx = self.np_random.integers(-9999, 9999)

        self.create_landers()

        for lander in self.landers:
            lander.ApplyForceToCenter(
            (
                self.np_random.uniform(-INITIAL_RANDOM, INITIAL_RANDOM),
                self.np_random.uniform(-INITIAL_RANDOM, INITIAL_RANDOM),
            ),
            True,
        )
        self.drawlist = self.legs + self.landers
        if self.render_mode == "human": self.render()

        self.agents = self.possible_agents[:]
        self._agent_selector = agent_selector(self.agents)
        self.agent_selection = self._agent_selector.next()
        self.rewards = {agent: 0 for agent in self.agents}
```

```
        self._cumulative_rewards = {agent: 0 for agent in self.agents}
        self.terminations = {agent: False for agent in self.agents}
        self.truncations = {agent: False for agent in self.agents}
        self.infos = {agent: {} for agent in self.agents}

        return self.observe(), None
    # End reset()


    def observe(self, agent=None):
        if not agent : agent = self.agent_selection
        id = self.agent_name_mapping[agent]
        lander = self.landers[id]
        pos = lander.position
        vel = lander.linearVelocity

        observation = [
            (pos.x - VIEWPORT_W/SCALE/2) / (VIEWPORT_W/SCALE/2),
            (pos.y - (self.helipad_y + LEG_DOWN/SCALE)) / (VIEWPORT_H/SCALE/2),
            vel.x * (VIEWPORT_W/SCALE/2) / FPS,
            vel.y * (VIEWPORT_H/SCALE/2) / FPS,
            lander.angle,
            20.0 * lander.angularVelocity / FPS,
            1.0 if self.legs[id*2+0].ground_contact else 0.0,
            1.0 if self.legs[id*2+1].ground_contact else 0.0,
        ]
        assert len(observation) == 8
        return observation

    def latest_reward_state(self, agent, state):
        reward = 0
        id = self.agent_name_mapping[agent]
        shaping = (
            -100 * np.sqrt(state[0] * state[0] + state[1] * state[1])
            - 100 * np.sqrt(state[2] * state[2] + state[3] * state[3])
            - 100 * abs(state[4])
            + 10 * state[6] + 10 * state[7]
        )   # And ten points for legs contact, the idea is if you
            # lose contact again after landing, you get negative reward
        if self.prev_shaping[id] is not None:
            reward = shaping - self.prev_shaping[id]
        self.prev_shaping[id] = shaping
        return reward


    def step(self, action):
        assert self.landers is not None, "You forgot to call reset()"
        if (
            self.terminations[self.agent_selection]
            or self.truncations[self.agent_selection]
        ):
            # If one agent has terminated this accepts a None action,
            # which otherwise errors, handles stepping to the next agent
            obs = self.observe(self.agent_selection)
            self.agent_selection = self._agent_selector.next()
            return obs, 0, True, False, {}

        # Update wind and apply to the lander
        if self.enable_wind and not any([l.ground_contact for l in self.legs]):
            # the function used for wind is tanh(sin(2 k x) + sin(pi k x)),
```

```
            # which is proven to never be periodic, k = 0.01
            wind_mag = (
                math.tanh(
                    math.sin(0.02 * self.wind_idx)
                    + (math.sin(math.pi * 0.01 * self.wind_idx))
                ) * self.wind_power
            )
            self.wind_idx += 1

            for lander in self.landers:
                lander.ApplyForceToCenter(
                    (wind_mag, 0.0),
                    True,
                )

            # the function used for torque is tanh(sin(2 k x) + sin(pi k x)),
            # which is proven to never be periodic, k = 0.01
            torque_mag = (
                math.tanh(
                    math.sin(0.02 * self.torque_idx)
                    + (math.sin(math.pi * 0.01 * self.torque_idx))
                ) * self.turbulence_power
            )
            self.torque_idx += 1

            for lander in self.landers:
                lander.ApplyTorque(
                    torque_mag,
                    True,
                )

        # For current agent:
        agent = self.agent_selection
        lander = self.landers[self.agent_name_mapping[agent]]
        # Check action validity
        if self.continuous:
            action = np.clip(action, -1, +1).astype(np.float64)
        else:
            assert self.action_spaces[agent].contains(
                action), f"{action!r} ({type(action)}) invalid "

        # Tip is the (X and Y) components of the rotation of the lander.
        tip = (math.sin(lander.angle), math.cos(lander.angle))

        # Side is the (-Y and X) components of the rotation of the lander.
        side = (-tip[1], tip[0])

        # Generate two random numbers between -1/SCALE and 1/SCALE.
        dispersion = [self.np_random.uniform(-1.0,+1.0)/SCALE for _ in range(2)]

        m_power = 0.0
        if (self.continuous and action[0] > 0.0) or (
            not self.continuous and action == 2
        ):
            # Main engine
            if self.continuous:
                m_power = (np.clip(action[0], 0.0, 1.0) + 1.0) * 0.5  # 0.5..1.0
                assert m_power >= 0.5 and m_power <= 1.0
            else:
                m_power = 1.0
```

```
599
600             # 4 is move a bit downwards, +-2 for randomness
601             # The components of the impulse to be applied by the main engine.
602             ox = (
603                 tip[0] * (MAIN_ENGINE_Y_LOCATION / SCALE + 2 * dispersion[0])
604                 + side[0] * dispersion[1]
605             )
606             oy = (
607                 -tip[1] * (MAIN_ENGINE_Y_LOCATION / SCALE + 2 * dispersion[0])
608                 - side[1] * dispersion[1]
609             )
610             impulse_pos = (lander.position[0] + ox, lander.position[1] + oy)
611             if self.render_mode is not None:
612                 # particles are just a decoration, with no physics impact,
613                 # so don't add them when not rendering
614                 p = self._create_particle(
615                     3.5,  # 3.5 is here to make particle speed adequate
616                     impulse_pos[0],
617                     impulse_pos[1],
618                     m_power,
619                 )
620                 p.ApplyLinearImpulse(
621                     (
622                         ox * MAIN_ENGINE_POWER * m_power,
623                         oy * MAIN_ENGINE_POWER * m_power,
624                     ),
625                     impulse_pos,
626                     True,
627                 )
628             lander.ApplyLinearImpulse(
629                 (-ox * MAIN_ENGINE_POWER * m_power,
630                  -oy * MAIN_ENGINE_POWER * m_power),
631                 impulse_pos,
632                 True,
633             )
634
635         s_power = 0.0
636         if (self.continuous and np.abs(action[1]) > 0.5) or (
637             not self.continuous and action in [1, 3]
638         ):
639             # Orientation/Side engines
640             if self.continuous:
641                 direction = np.sign(action[1])
642                 s_power = np.clip(np.abs(action[1]), 0.5, 1.0)
643                 assert s_power >= 0.5 and s_power <= 1.0
644             else:
645                 # action = 1 is left, action = 3 is right
646                 direction = action - 2
647                 s_power = 1.0
648
649             # The components of the impulse to be applied by the side engines.
650             ox = tip[0] * dispersion[0] + side[0] * (
651                 3 * dispersion[1] + direction * SIDE_ENGINE_AWAY / SCALE
652             )
653             oy = -tip[1] * dispersion[0] - side[1] * (
654                 3 * dispersion[1] + direction * SIDE_ENGINE_AWAY / SCALE
655             )
656
657             # The constant 17 is presumably meant to be SIDE_ENGINE_HEIGHT.
658             # However, SIDE_ENGINE_HEIGHT is defined as 14, causing the
```

```
            # position of the thrust on the body of the lander to change,
            # depending on the orientation of the lander. This results in
            # an orientation dependent torque being applied to the lander.

            impulse_pos = (
                lander.position[0] + ox - tip[0] * 17 / SCALE,
                lander.position[1] + oy + tip[1] * SIDE_ENGINE_HEIGHT / SCALE,
            )
            if self.render_mode is not None:
                # particles are just decoration, with no impact on the physics,
                # so don't add them when not rendering
                p = self._create_particle(0.7, impulse_pos[0],
                                          impulse_pos[1], s_power)
                p.ApplyLinearImpulse(
                    (ox * SIDE_ENGINE_POWER * s_power,
                     oy * SIDE_ENGINE_POWER * s_power,),
                    impulse_pos,
                    True,
                )
            lander.ApplyLinearImpulse(
                (-ox * SIDE_ENGINE_POWER * s_power,
                 -oy * SIDE_ENGINE_POWER * s_power),
                impulse_pos,
                True,
            )

        # Update Positions
        self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)

        observation = self.observe(agent)
        reward = self.latest_reward_state(agent, observation)
        reward -= m_power * 0.30 # Deduct cost of fuel
        reward -= s_power * 0.03 # Deduct cost of fuel

        if self.game_over or abs(observation[0]) >= 1.0:
            self.terminations[self.agent_selection] = True
            reward = -100
        if not lander.awake:
            self.terminations[self.agent_selection] = True
            reward = +100
        terminated = self.terminations[self.agent_selection]

        self.rewards[agent] = reward
        self._accumulate_rewards()

        self.agent_selection = self._agent_selector.next()

        if self.render_mode == "human":
            self.render()
        # truncation=False as the time limit is handled by the `TimeLimit`
        return observation, reward, terminated, False, {}


class Parallel_Env(SequentialEnv):
    def observe(self):
        obs_list = []
        for i,lander in enumerate(self.landers):
            pos = lander.position
            vel = lander.linearVelocity
            obs = [
```

```
                (pos.x - VIEWPORT_W/SCALE/2) / (VIEWPORT_W/SCALE/2),
                (pos.y - (self.helipad_y + LEG_DOWN/SCALE)) / (VIEWPORT_H/SCALE/2),
                vel.x * (VIEWPORT_W/SCALE/2) / FPS,
                vel.y * (VIEWPORT_H/SCALE/2) / FPS,
                lander.angle,
                20.0 * lander.angularVelocity / FPS,
                1.0 if self.legs[i*2+0].ground_contact else 0.0,
                1.0 if self.legs[i*2+1].ground_contact else 0.0,
            ]
            assert len(obs) == 8
            obs_list.append(obs)
        return dict(zip(self.possible_agents,obs_list))

    def last(self):
        return self.observe()

    def step(self, action_list):
        self._clear_rewards()
        assert self.landers is not None, "You forgot to call reset()"

        # Update wind and apply to the lander
        if self.enable_wind and not any([l.ground_contact for l in self.legs]):
            # the function used for wind is tanh(sin(2 k x) + sin(pi k x)),
            # which is proven to never be periodic, k = 0.01
            wind_mag = (
                math.tanh(
                    math.sin(0.02 * self.wind_idx)
                    + (math.sin(math.pi * 0.01 * self.wind_idx))
                ) * self.wind_power
            )
            self.wind_idx += 1

            for lander in self.landers:
                lander.ApplyForceToCenter(
                    (wind_mag, 0.0),
                    True,
                )
            # the function used for torque is tanh(sin(2 k x) + sin(pi k x)),
            # which is proven to never be periodic, k = 0.01
            torque_mag = (
                math.tanh(
                    math.sin(0.02 * self.torque_idx)
                    + (math.sin(math.pi * 0.01 * self.torque_idx))
                ) * self.turbulence_power
            )
            self.torque_idx += 1

            for lander in self.landers:
                lander.ApplyTorque(
                    torque_mag,
                    True,
                )

        #for agent in self.agents:
        for agent,action in action_list.items():
            # For current agent:
            lander = self.landers[self.agent_name_mapping[agent]]
            # Check action validity
            if self.continuous:
                action = np.clip(action, -1, +1).astype(np.float64)
```

```
        else:
            assert self.action_spaces[agent].contains(
                action), f"{action!r} ({type(action)}) invalid "

        # Tip is the (X and Y) components of the rotation of the lander.
        tip = (math.sin(lander.angle), math.cos(lander.angle))

        # Side is the (-Y and X) components of the rotation of the lander.
        side = (-tip[1], tip[0])

        # Generate two random numbers between -1/SCALE and 1/SCALE.
        dispersion = [self.np_random.uniform(-1.0,+1.0)/SCALE for _ in range(2)]

        m_power = 0.0
        if (self.continuous and action[0] > 0.0) or (
            not self.continuous and action == 2
        ):
            # Main engine
            if self.continuous:
                m_power = (np.clip(action[0], 0.0, 1.0) + 1.0) * 0.5  # 0.5..1.0
                assert m_power >= 0.5 and m_power <= 1.0
            else:
                m_power = 1.0

            # 4 is move a bit downwards, +-2 for randomness
            # The components of the impulse to be applied by the main engine.
            ox = (
                tip[0] * (MAIN_ENGINE_Y_LOCATION / SCALE + 2 * dispersion[0])
                + side[0] * dispersion[1]
            )
            oy = (
                -tip[1] * (MAIN_ENGINE_Y_LOCATION / SCALE + 2 * dispersion[0])
                - side[1] * dispersion[1]
            )
            impulse_pos = (lander.position[0] + ox, lander.position[1] + oy)
            if self.render_mode is not None:
                # particles are just a decoration, with no physics impact,
                # so don't add them when not rendering
                p = self._create_particle(
                    3.5,  # 3.5 is here to make particle speed adequate
                    impulse_pos[0],
                    impulse_pos[1],
                    m_power,
                )
                p.ApplyLinearImpulse(
                    (
                        ox * MAIN_ENGINE_POWER * m_power,
                        oy * MAIN_ENGINE_POWER * m_power,
                    ),
                    impulse_pos,
                    True,
                )
            lander.ApplyLinearImpulse(
                (-ox * MAIN_ENGINE_POWER * m_power,
                -oy * MAIN_ENGINE_POWER * m_power),
                impulse_pos,
                True,
            )

        s_power = 0.0
```

```
            if (self.continuous and np.abs(action[1]) > 0.5) or (
                not self.continuous and action in [1, 3]
            ):
                # Orientation/Side engines
                if self.continuous:
                    direction = np.sign(action[1])
                    s_power = np.clip(np.abs(action[1]), 0.5, 1.0)
                    assert s_power >= 0.5 and s_power <= 1.0
                else:
                    # action = 1 is left, action = 3 is right
                    direction = action - 2
                    s_power = 1.0

                # The components of the impulse to be applied by the side engines.
                ox = tip[0] * dispersion[0] + side[0] * (
                    3 * dispersion[1] + direction * SIDE_ENGINE_AWAY / SCALE
                )
                oy = -tip[1] * dispersion[0] - side[1] * (
                    3 * dispersion[1] + direction * SIDE_ENGINE_AWAY / SCALE
                )

                # The constant 17 is presumably meant to be SIDE_ENGINE_HEIGHT.
                # However, SIDE_ENGINE_HEIGHT is defined as 14, causing the
                # position of the thrust on the body of the lander to change,
                # depending on the orientation of the lander. This results in
                # an orientation dependent torque being applied to the lander.

                impulse_pos = (
                    lander.position[0] + ox - tip[0] * 17 / SCALE,
                    lander.position[1] + oy + tip[1] * SIDE_ENGINE_HEIGHT / SCALE,
                )
                if self.render_mode is not None:
                    # particles are just decoration, with no impact on the physics,
                    # so don't add them when not rendering
                    p = self._create_particle(0.7, impulse_pos[0],
                                              impulse_pos[1], s_power)
                    p.ApplyLinearImpulse(
                        (ox * SIDE_ENGINE_POWER * s_power,
                        oy * SIDE_ENGINE_POWER * s_power,),
                        impulse_pos,
                        True,
                    )
                lander.ApplyLinearImpulse(
                    (-ox * SIDE_ENGINE_POWER * s_power,
                    -oy * SIDE_ENGINE_POWER * s_power),
                    impulse_pos,
                    True,
                )
                self.rewards[agent] -= m_power * 0.30 # Deduct cost of fuel
                self.rewards[agent] -= s_power * 0.03 # Deduct cost of fuel
        """ End Agent Loop """

        # Update Positions
        self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)

        obs_list = self.observe()

        for agent in self.agents:
            obs = obs_list[agent]
            lander = self.landers[self.agent_name_mapping[agent]]
```

```
            self.rewards[agent] += self.latest_reward_state(agent, obs)
            if self.game_over or abs(obs[0]) >= 1.0:
                self.terminations[agent] = True
                self.rewards[agent] -= 100
            if not lander.awake:
                self.terminations[agent] = True
                self.rewards[agent] += 100

        self._accumulate_rewards()

        if self.render_mode == "human":
            self.render()
        # truncation=False as the time limit is handled by the `TimeLimit`
        return obs_list, self.rewards, self.terminations, self.truncations, {}


def heuristic(env, s):
    """
    The heuristic for
    1. Testing
    2. Demonstration rollout.

    Args:
        env: The environment
        s (list): The state. Attributes:
            s[0] is the horizontal coordinate
            s[1] is the vertical coordinate
            s[2] is the horizontal speed
            s[3] is the vertical speed
            s[4] is the angle
            s[5] is the angular speed
            s[6] 1 if first leg has contact, else 0
            s[7] 1 if second leg has contact, else 0

    Returns:
         a: The heuristic to be fed into the step function defined above to
            determine the next step and reward.
    """

    angle_targ = s[0] * 0.5 + s[2] * 1.0  # angle should point towards center
    if angle_targ > 0.4:
        angle_targ = 0.4  # more than 0.4 radians (22 degrees) is bad
    if angle_targ < -0.4:
        angle_targ = -0.4
    hover_targ = 0.55 * np.abs(
        s[0]
    )  # target y should be proportional to horizontal offset

    angle_todo = (angle_targ - s[4]) * 0.5 - (s[5]) * 1.0
    hover_todo = (hover_targ - s[1]) * 0.5 - (s[3]) * 0.5

    if s[6] or s[7]:  # legs have contact
        angle_todo = 0
        hover_todo = (
            -(s[3]) * 0.5
        )  # override to reduce fall speed, that's all we need after contact

    if env.unwrapped.continuous:
        a = np.array([hover_todo * 20 - 1, -angle_todo * 20])
        a = np.clip(a, -1, +1)
```

```
    else:
        a = 0
        if hover_todo > np.abs(angle_todo) and hover_todo > 0.05:
            a = 2
        elif angle_todo < -0.05:
            a = 3
        elif angle_todo > +0.05:
            a = 1
    return a

def demo_heuristic_lander(env, reps=1, seed=None, render=False):
    total_reward = 0
    steps = 0
    for _ in range(reps):
        s = env.reset(seed=seed)
        while True:
            s = env.last()[0]
            a = heuristic(env, s)
            s, r, terminated, truncated, info = env.step(a)
            total_reward += r

            if render:
                still_open = env.render()
                if still_open is False:
                    break

            if steps % 20 == 0 or terminated or truncated:
                print("observations:", " ".join([f"{x:+0.2f}" for x in s]))
                print(f"step {steps} total_reward {total_reward:+0.2f}")
                print(env.terminations.values())
            steps += 1
            # if terminated or truncated:
            if all(env.terminations.values()):
                break
    if render:
        env.close()
    return total_reward

def demo_parallel_heuristic(env, reps=1, seed=None, render=False):
    total_reward = 0
    steps = 0
    for _ in range(reps):
        _ = env.reset(seed=seed)
        obs = env.last()
        while True:
            actions = {agent: heuristic(env, obs[agent]) for agent in env.agents}
            obs, r, terminateds, truncateds, info = env.step(actions)
            total_reward += sum(r.values())

            if render:
                still_open = env.render()
                if still_open is False:
                    break

            if steps % 20 == 0 or all(terminateds) or all(truncateds):
                for agent in env.agents:
                    print(f"{agent} observations:", " ".join([f"{x:+0.2f}" for x in obs[ag
                print(f"step {steps} total_reward {total_reward:+0.2f}")
                print(env.terminations.values())
            steps += 1
```

```
            # if terminated or truncated:
            if all(env.terminations.values()):
                break
    if render:
        env.close()
    return total_reward

def parallel_env(**kwargs):
    return Parallel_Env(**kwargs)

def sequential_env(**kwargs):
    return SequentialEnv(**kwargs)

def env(**kwargs):
    return Parallel_Env(**kwargs)

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description=
                                     'A multi-agent version of Lunar Lander')
    parser.add_argument('-s', '--sequential', action="store_true",
                        help='Iterate as AEC environment')
    parser.add_argument('-n', '--num_landers', type=int, default=2,
                        help='number of landers')
    parser.add_argument('-d', '--demo_iters', type=int, default=1,
                        help='number of landers')
    args = parser.parse_args()

    if args.sequential:
        _env = sequential_env(render_mode="human", num_landers=args.num_landers)
        demo_heuristic_lander(_env, reps=args.demo_iters, render=True)
    else:
        _env = parallel_env(render_mode="human", num_landers=args.num_landers)
        demo_parallel_heuristic(_env, reps=args.demo_iters, render=True)
```