Supplemental Material for Submission 586
Deep Reinforcement Learning for Active Wake Control

GRIGORY NEUSTROEV, Delft University of Technology, the Netherlands
SYTZE P. E. ANDRINGA, Delft University of Technology, the Netherlands
REMCO A. VERZIJLBERGH, Delft University of Technology & Whiffle, the Netherlands
MATHIJS M. DE WEERDT, Delft University of Technology, the Netherlands

ACM Reference Format:
Grigory Neustroev, Sytze P. E. Andringa, Remco A. Verzijlbergh, and Mathijs M. de Weerdt. 2022. Supplemental Material for Submission 586 Deep Reinforcement Learning for Active Wake Control. In *Proc. of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2022), Online, May 9–13, 2022,* IFAAMAS, 3 pages.

1 WINDFARMENV API

Our wind farm environment follows the OpenAI *Gym* convention. It is implemented as a class `WindFarmEnv` derived from the base `Env` class of *Gym*, and has the following methods:

- `step(action)`: takes the agent's `action`, runs a single step of the environment's dynamics, and returns a tuple of four elements:
  - observation,
  - reward,
  - done: in *Gym* signifies terminal states, currently always `False`,
  - info: diagnostic information, currently empty.
- `reset()`: resets the environment to the initial state, including the atmospheric simulation process.
- `render(mode)`: renders a single frame of the environment; supports two modes: `'human'` to play the frame (usually called in a loop and results in an animation) and `'rgb_array'` (for recording).
- `close()`: finalizes the environment.
- `seed()`: sets the seed for the random number generator.

The constructor `WindFarmEnv(...)` supports the following parameters. All of them are optional, and default values will be used if no value is provided.

- `seed`: random seed;
- `floris`: either an existing `floris_interface` object or a path to a JSON file with FLORIS configuration required to instantiate a FLORIS model;
- `time_delta`: time interval Δt (in sec) between time steps;
- `turbine_layout`, `mast_layout`: coordinates of the turbines and meteorological masts;
- `observe_yaws`: if `True`, yaws are added to the state vector;
- `lidar_turbines`: either `'all'` or a list of turbines which have nacelle-mounted lidars;
- `lidar_range`: distance in meters between the rotor and the lidar measurement point;
- `farm_observations`, `mast_observations`,
  `lidar_observations`: lists of atmospheric conditions observed at each level;
- `normalize_observations`: if `True`, the observations will be rescaled to $[0, 1]$;
- `observation_boundaries`: used for rescaling the observations;

- `perturbed_observations`: indices of observations to add Gaussian noise to;
- `perturbation_scale`: standard deviation of the noise relative to their scale;
- `max_angular_velocity`: maximum turbine rotation $\omega_{\max}$ per time step;
- `desired_yaw_boundaries`: yaw boundaries relative to the wind, $\gamma_{\min}$ and $\gamma_{\max}$;
- `action_representation`: `'yaw'`, `'absolute'`, or `'wind'`;
- `wind_process`: the stochastic process governing the transitions;
- `random_reset`: if `True`, upon reset the environment is set to a random yaws.

The use of `WindFarmEnv` is similar to other *Gym* environments. For example, the following code will show an animation of the environment with the default settings and random actions being performed for 1000 steps.

```python
from wind_farm_gym import WindFarmEnv

env = WindFarmEnv()

observation = env.reset()
for i in range(1000):
  action = env.action_space.sample()
  observation, reward, _, _ = env.step(action)
  env.render()
```

Listing 1. Minimal example

1.1 WindProcess API

State transitions of `WindFarmEnv` are generated by an object of class `WindProcess`. Users can create their own implementations, as long as they implement the following API:

- `step()`: returns a dictionary of atmospheric conditions for the next step of the environment, for example, `{'wind_direction': 270.0, 'wind_speed': 9.5}`.
- `reset()`.
- `close()`: finalizes the process; for example, if the process uses lazy reading from a file, this method should close the file.

This process does not have to generate the transitions randomly, so the seeding method is not mandatory.

As part of our environment, we include three classes of wind data processes:

- MVOU-driven transitions. The user can initialize arbitrary multivariate processes, as long as they provide the required vectors and matrices and give a list of measurements to map the data to.
- Transitions from a `.csv` file. If the user has access to historical data at a sufficiently fine resolution, they can use it to generate each time step of the simulation. This can be useful for wind energy researchers who may have access to better wind models than a simple MVOU process we provide.
- Gaussian noise. This can be achieved with MVOU by using zero drift and an identity diffusion matrix, but we add it separately for more efficient computations, as Gaussian noise is used in observation perturbations.

2 EXPERIMENT IMPLEMENTATION DETAILS

The hyperparameters used by each agent in the benchmarks are listed in Table 1. The second experiment uses the same parameters unless explicitly listed. After the first experiment, the learning rates of SAC were additionally tuned for the second experiment using a grid search.

Table 1. Hyperparameters of the deep reinforcement learning agents

	parameter	I	II	notes
	discounting factor	0.99		
sampling	batch size	128		
	size of replay buffer	10^5		
	start learning at step	4321	7201	after the first evaluation
actor	learning rate	10^{-3}	10^{-5}	
	layers	10^{-2}	10^{-4}	
	neurons per layer	128		
	activations	ReLU		
critic	learning rate	0.01		
	layers	2		
	neurons per layer	128		
	activations	ReLU		
target updates	Polyak tau	0.05		
	frequency	60		
TD3	policy noise	0.2		only in training
	policy update frequency	60		
	noise clipped at	±0.5		
	gradient norm clipped at	±0.5		
SAC	initial alpha	1.0		using autotuning method
	alpha learning rate	10^{-2}	10^{-4}	