Learning to Fly—a Gym Environment with PyBullet Physics for Reinforcement Learning of Multi-agent Quadcopter Control

Jacopo Panerati,[1,2] Hehui Zheng,[3] SiQi Zhou,[1,2] James Xu,[1] Amanda Prorok,[3] and Angela P. Schoellig[1,2]

***Abstract*— Robotic simulators are crucial for academic research and education as well as the development of safety-critical applications. Reinforcement learning *environments*—simple simulations coupled with a problem specification in the form of a reward function—are also important to standardize the development (and benchmarking) of learning algorithms. Yet, full-scale simulators typically lack portability and parallelizability. Vice versa, many reinforcement learning environments trade-off realism for high sample throughputs in toy-like problems. While public data sets have greatly benefited deep learning and computer vision, we still lack the software tools to simultaneously develop—and fairly compare—control theory and reinforcement learning approaches. In this paper, we propose an open-source OpenAI Gym-like environment for multiple quadcopters based on the Bullet physics engine. Its multi-agent and vision-based reinforcement learning interfaces, as well as the support of realistic collisions and aerodynamic effects, make it, to the best of our knowledge, a first of its kind. We demonstrate its use through several examples, either for control (trajectory tracking with PID control, multi-robot flight with downwash, etc.) or reinforcement learning (single and multi-agent stabilization tasks), hoping to inspire future research that combines control theory and machine learning.**

I. Introduction

Over the last decade, the progress of machine learning—and deep learning specifically—has revolutionized computer science by obtaining (or surpassing) human performance in several tasks, including image recognition and game playing [1]. New algorithms, coupled with shared benchmarks and data sets have greatly contributed to the advancement of multiple fields (e.g., as the KITTI suite [2] did for computer vision in robotics). While reinforcement learning (RL) looks as a very appealing solution to bridge the gap between control theory and deep learning, we are still in the infancy of the creation of tools for the development of realistic continuous control applications through deep RL [3].

As automation becomes more pervasive —from healthcare to aerospace, from package delivery to disaster recovery—better tools to design robotic applications are also required. Simulations are an indispensable step in the design of both robots and their control approaches [4], especially so when building a prototype is expensive and/or safety (of the hardware and its surroundings) is a concern. Besides platform-specific and proprietary software, many of today's open-source robotic simulators are based on ROS's *plumbing* and engines like Gazebo and Webots. While these solutions can leverage a host of existing plugins, their limited portability can hinder typical machine learning workflows, based on remote, highly parallel, computing cluster execution.

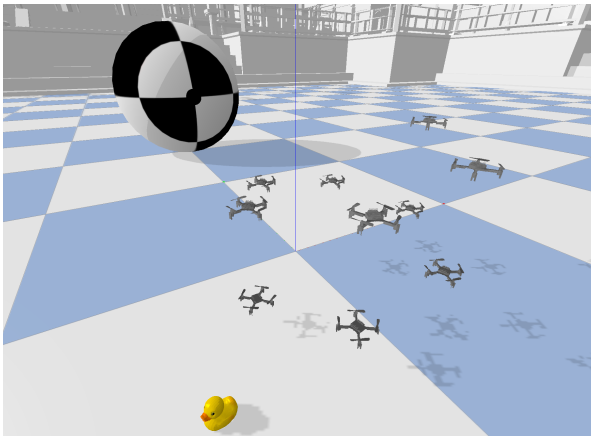

Fig. 1. Rendering of a `gym-pybullet-drones` simulation with 10 Crazyflie 2.x on a circular trajectory and a rubber duck for scale.

In an attempt to standardize and foster deep RL research, over the last few years, RL *environments* have multiplied. OpenAI's *Gym* [5] emerged as a standard that comprises *(i)* a suite of benchmark problems as well as *(ii)* an API for the definition of new environments. Because of the inherent similarities between the decision-making loops of control theory and RL, many popular *environments* are inspired by control tasks (e.g. the balancing of a pole on a cart). However, because deep RL algorithms often rely on large amounts of data, some of these environments trade-off realism for high sample throughputs. A reason for concern is that developing—and benchmarking—algorithms on environments that are not necessarily representative of practical scenarios might curb the progress of RL in robotics [6].

With this work, we want to provide both the robotics and machine learning (ML) communities with a compact, open-source *Gym*-style environment[a] that supports the definition of multiple learning tasks (multi-agent RL, vision-based RL, etc.) on a practical robotic application: the control of one or more nanoquadcopters. The software[bc] provided with this paper, `gym-pybullet-drones`, can help both roboticists and ML engineers to develop end-to-end quadcopter control with model-free or model-based RL. The main features of `gym-pybullet-drones` are:

1) *Realism*: support for realistic collisions, aerodynamics effects, and extensible dynamics *via* Bullet Physics [10].

[1]Jacopo Panerati, SiQi Zhou, James Xu, and Angela P. Schoellig are with the Dynamic Systems Lab, Institute for Aerospace Studies, University of Toronto, Canada, e-mails: {name.lastname}@utoronto.ca; and the [2]Vector Institute for Artificial Intelligence in Toronto. [3]Hehui Zheng and Amanda Prorok are with the the Prorok Lab and the Department of Computer Science and Technology, University of Cambridge, Cambridge, United Kingdom, e-mails: {hz337, asp45}@cam.ac.uk.

[a]Video: `www.tiny.cc/ucsqtz`
[b]Web: `utiasdsl.github.io/gym-pybullet-drones/`
[c]Code: `github.com/utiasDSL/gym-pybullet-drones`

TABLE I

FEATURE COMPARISON BETWEEN THIS WORK AND RECENT QUADCOPTER SIMULATORS WITH A FOCUS ON RL OR THE CRAZYFLIE 2.X

	Physics Engine	Rendering Engine	Language	Synchro./Steppable Physics & Rendering	RGB, Depth, and Segmentation Views	Multiple Vehicles	*Gym* API	Multi-agent *Gym*-like API
This work	PyBullet	OpenGL3†	Python	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Flightmare [7]	*Ad hoc*	Unity	C++	**Yes**	**Yes**	**Yes**	W/o Vision	No
AirSim [8]	PhysX¶	UE4	C++	No	**Yes**	**Yes**	No	No
CrazyS [9]	Gazebo§	OGRE	C++	**Yes**	No Segmentation	No	No	No

† or TinyRenderer ¶ or FastPhysicsEngine § ODE, Bullet, DART, or Simbody

2) *RL Flexibility*: availability of *Gym*-style environments for both vision-based RL and multi-agent RL—simultaneously, if desired.
3) *Parallelizability*: multiple environments can be easily executed, with a GUI or headless, with or without a GPU, with minimal installation requirements.
4) *Ease-of-use*: pre-implemented PID control, as well as Stable Baselines3 [11] and RLlib workflows [12].

Section II of this paper reviews similar simulation environments for RL, in general, and quadcopters, specifically. Section III details the inner working and programming interfaces of our *Gym* environment. In Sections IV and V, we analyze its computing performance and provide control and learning use cases. Section VI suggests the possible extensions of this work. Finally, Section VII concludes the paper.

II. RELATED WORK

Several reinforcement learning environments and quadcopter simulators are already available to the community, offering different features and capabilities. Here, we briefly discuss *(i)* the current landscape of RL environments, *(ii)* the learning interfaces of existing quadcopter simulators, and *(iii)* how they compare to `gym-pybullet-drones`.

A. Reinforcement Learning Environments

The OpenAI *Gym* toolkit [5] was created in 2016 to address the lack of standardization among the benchmark problems used in reinforcement learning research and, within five years, it was cited by over 2000 publications. Besides the standard API adopted in this work, it comprises multiple problem sets. Some of the simplest, "Classical control" and "Box2D", are two-dimensional, often discrete action problems—e.g., the swing-up of a pendulum. The more complex problems, "Robotics" and "MuJoCo", include continuous control of robotic arms and legged robots in three-dimensions (Swimmer, Hopper, HalfCheetah, etc.) that are based on the proprietary MuJoCo physics engine [13].

MuJoCo's physics engine also powers DeepMind's `dm_control` [14]. While `dm_control`'s environments do not expose the same API as *Gym*, they are very similarly structured and DeepMind's suite includes many of the same articulated-body locomotion and manipulation tasks. However, because of smoothing around the contacts and other simplifications, even locomotion policies trained successfully with these environments do not necessarily exhibit gaits that would easily transfer to physical robots [3].

The need for MuJoCo's licensing also led to the development and adoption of open-source alternatives such as Georgia Tech/CMU's DART and Google's Bullet Physics [10] (with its Python binding, PyBullet). Open-source Bullet-based re-implementations of the control and locomotion tasks in [5] are also provided in `pybullet-gym`. Community-contributed *Gym* environments like `gym-minigird` [15]—a collection of 2D grid environments—were used by over 30 publications between 2018 and 2021.

Both OpenAI and Google Research have made recent strides to include safety requirements and real-world uncertainties in their control and legged locomotion RL benchmarks with `safety-gym` [16] and `realworldrl_suite` [17], respectively.

One of the most popular *Gym* environment for quadcopters is `gymfc` [18]. While having a strong focus on the transferability to real hardware, the work in [18] only addresses the learning of an attitude control loop that exceeds the performance of a PID implementation, using Gazebo simulations. Work similar to [18], training a neural network in simulation for the sim2real stabilization of a Crazyflie 2.x (the same quadcopter model used here), is presented in [19]. To the best of our knowledge, `gym-pybullet-drones` is the first general purpose multi-agent *Gym* environment for quadcopters.

B. Quadcopter Simulators

RotorS [20] is a popular quadcopter simulator based on ROS and Gazebo. It includes multiple AscTec multirotor models and simulated sensors (IMU, etc.). However, it does not come with ready-to-use RL interfaces and its dependency on Gazebo can make it ill-advised for parallel execution or vision-based learning applications. CrazyS [9] is an extension of RotorS that is specifically targeted to the Bitcraze Crazyflie 2.x nanoquadcopter. Due to its accessibility and popularity in research, we also chose the Crazyflie to be the default quadcopter model in `gym-pybullet-drones`. However, for RL applications, CrazyS suffers from the same limitations as RotorS.

Microsoft's AirSim [8] is one of the best known simulators supporting multiple vehicles—car and quadcopters—and photorealistic rendering through Unreal Engine 4. While being an excellent choice for the development of self-driving applications, its elevated computational requirements and overly simplified collisions—using FastPhysicsEngine in multirotor mode—make it less than ideal for learning control. AirSim also lacks a native *Gym* interface, yet wrappers for velocity input control have been proposed [21].

The most recent and closely related work to ours is ETH's Unity-based Flightmare [7]. This simulator was created to simultaneously provide photorealistic rendering and very fast, highly parallel dynamics. Flightmare also implements *Gym*'s API and it includes a single agent RL workflow. Unlike `gym-pybullet-drones`, however, Flighmare does not include a *Gym* with vision-based observations nor one compatible with multi-agent reinforcement learning (MARL).

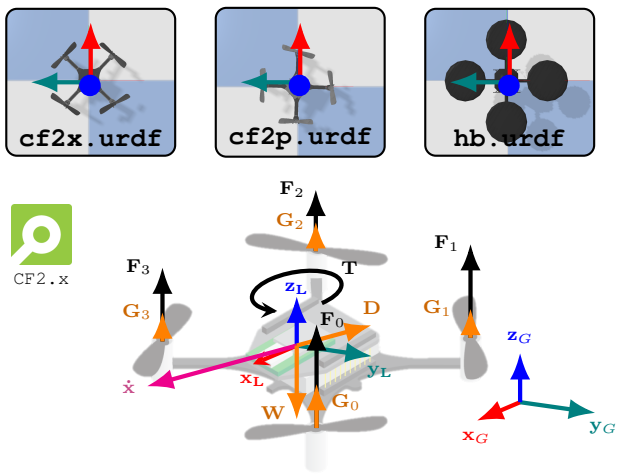

Fig. 2. The three—1 in × and 2 in + configuration—quadcopter models in `gym-pybullet-drones` (top) and the forces and torques acting on each vehicle, as modeled in Section III-C (bottom).

Table I summarizes the main features of CrazyS, AirSim, Flightmare, and `gym-pybullet-drones`.

III. Methods

To explain how `gym-pybullet-drones` works, one needs to understand *(i)* how its dynamics evolve (Subsections III-A to III-C), *(ii)* which types of observations, including vision and multi-agent ones, can be extracted from it (Subsection III-D), *(iii)* what commands one can issue (Subsection III-E), and *(iv)* which learning and control workflows we built on top of it (Subsections III-F and III-G).

A. Gym Environment Classes

OpenAI's *Gym* toolkit was introduced to standardize the development of RL problems and algorithms in Python. As in a standard Markov decision process (MDP) framework, a generic *environment* class receives an *action*, uses it to update (*step*) its internal state, and returns a new state (*observation*) paired with a corresponding *reward*. This, of course, is akin to a simple feedback loop in which a controller feeds an input signal to a plant to receive a (possibly noisy) output measurement. An OpenAI *Gym* environment also exposes additional information about whether its latest state is terminal (*done*) and other standard APIs to *(i) reset* it between episodes and *(ii)* query it about the domains (*spaces*) of its actions and observations.

B. Bullet Physics

Physics engines are particularly appealing to researchers working on both robotics and ML because they *(i)* expedite the development and test of new applications for the former, while *(ii)* yielding large data sets for the latter [4]. The work in this paper is based on the open-source Bullet Physics engine [10]. Our choice was motivated by its collision management system, the availability of both CPU- and GPU-based rendering, the compatibility with the Unified Robot Description Format (URDF), and its steppable physics—allowing to extract synchronized rendering and kinematics, as well as to control, at arbitrary frequencies.

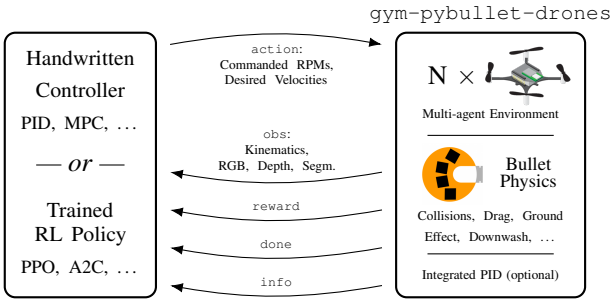

Fig. 3. Schematics of the handed over input parameters (`action`) and yielded return values (`obs`, `reward`, `done`, and `info`) by every call to the `step` method of a `gym-pybullet-drones` environment.

C. Quadcopter Dynamics

We use PyBullet to model the forces and torques acting on each quadcopter in our *Gym* and leverage the physics engine to compute and update the kinematics of all vehicles.

1) Quadcopter models: The default quadcopter model in `gym-pybullet-drones` is the Bitcraze Crazyflie 2.x. Its popularity and availability meant we could leverage both a wealth of system identification work [22]–[24] and real-world experiments to pick the parameters used in this section. Its arm length L, mass m, inertial properties $\mathbf{J}$, physical constants, and a convex collision shape are described through separate URDF files for the × and + configurations (see Figure 2). We also provide the URDF for a generic, larger quadcopter based on the Hummingbird described in [25].

2) PyBullet-based Physics Update: First, PyBullet let us set the gravity acceleration g and the physics stepping frequency (which can be much finer grained than the control frequency at which the *Gym* steps). Besides the physical properties and constants, PyBullet uses the URDF information to load a CAD model of the quadcopter. The forces F_i's applied to each of the 4 motors and the torque T induced around the drone's z-axis are proportional to the squared motor speeds P_i's in RPMs. These are linearly related to the input PWMs and we assume we can control them near-instantaneously [25]:

$$F_i = k_F \cdot P_i^2, \qquad T = \sum_{i=0}^{3} (-1)^{i+1} k_T \cdot P_i^2, \tag{1}$$

where k_F and k_T are constants.

a) Explicit Python Dynamics Update: As an alternative implementation, we also provide an explicit Python update that is not based on Bullet. This can be used for comparison, debugging, or the development of *ad hoc* dynamics [7]. In this case, the linear acceleration in the global frame $\ddot{\mathbf{x}}$ and change in the turn rates in the local frame $\dot{\boldsymbol{\psi}}$ are computed as follows:

$$\ddot{\mathbf{x}} = \left(\mathbf{R} \cdot [0, 0, k_F \textstyle\sum_{i=0}^{3} P_i^2] - [0, 0, mg]\right) m^{-1}, \tag{2}$$

$$\dot{\boldsymbol{\psi}} = \mathbf{J}^{-1} \left(\kappa_{\times}(L, k_F, k_T, [P_0^2, P_1^2, P_2^2, P_3^2]) - \boldsymbol{\psi} \times (\mathbf{J}\boldsymbol{\psi})\right), \tag{3}$$

where $\mathbf{R}$ is the rotation matrix of the quadcopter and $\kappa_{\times}$, κ_{+} are functions to compute the torques induced in the local frame by the motor speeds, for the × and + configuration.

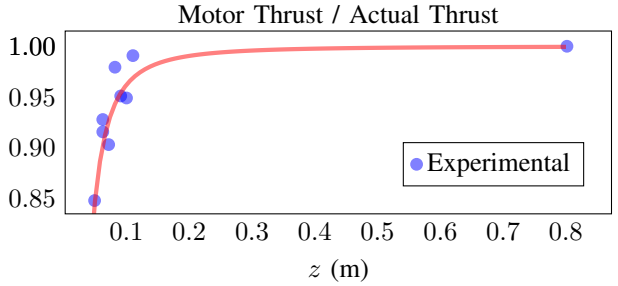

Fig. 4. The motor thrust-to-actual thrust ratio corresponds to the percentage of thrust actually provided by the motors as a quadcopter hovers at different altitudes in z. We used the experimental data in blue to fit the coefficient k_G in (5), yielding the ground effect profile plotted in red.

3) Aerodynamic Effects: While the model presented in (1) captures simple quadcopter dynamics, flying in a medium, in the proximity of the ground, or near other vehicles can result in additional aerodynamic effects (Figure 2). PyBullet allows to model these separately and use them jointly.

a) Drag: The spinning propellers of a quadcopter produce drag $\mathbf{D}$, a force acting in the direction opposite to the one of motion. Our modeling is based on [23] and it states that the air resistance is proportional to the quadcopter velocity $\dot{\mathbf{x}}$, the angular velocities of the rotors, and a matrix of coefficients $\mathbf{k_D}$ experimentally derived in [23]:

$$\mathbf{D} = -\mathbf{k_D}\left(\sum_{i=0}^{3}\frac{2\pi P_i}{60}\right)\dot{\mathbf{x}}. \tag{4}$$

b) Ground Effect: When hovering at a very low altitude, a quadcopter is subject to an increased thrust caused by the interaction of the propellers' airflow with the surface, i.e., the *ground effect*. Based on [26] and real-world experiments with Crazyflie hardware (see Figure 4), we model contributions G_i's for each motor that are proportional to the propellers' radius r_P, speeds P_i's, altitudes h_i's, and a constant k_G:

$$G_i = k_G k_F \left(\frac{r_P}{4h_i}\right)^2 P_i^2. \tag{5}$$

c) Downwash: When two quadcopters cross paths at different altitudes, the downwash effect causes a reduction in the lift of the bottom one. For simplicity, we model it as a single contribution applied to the center of mass of the quadcopter whose module W depends on the distances in x, y, and z between the two vehicles (δ_x, δ_y, δ_z) and constants k_{D_1}, k_{D_2}, k_{D_3} that we identified experimentally:

$$W = k_{D_1}\left(\frac{r_P}{4\delta_z}\right)^2 exp\left(-\frac{1}{2}\left(\frac{\sqrt{\delta_x^2+\delta_y^2}}{k_{D_2}\delta_z + k_{D_3}}\right)^2\right). \tag{6}$$

Figure 9 compares a flight simulation using this model with data from a real-world flight experiment.

D. Observation Spaces

Every time we advance `gym-pybullet-drones` by one step—which might include multiple steps of the physics engine—we receive an observation vector. In our code base, we provide several implementations. Yet, they all include the following kinematic information: a dictionary whose keys are drone indices $n \in [0..N]$ and values contain positions $\mathbf{x}_n = [x, y, z]_n$'s, quaternions $\mathbf{q}_n$'s, rolls r_n, pitches p_n, and yaws j_n's, linear $\dot{\mathbf{x}}_n$, and angular velocities $\boldsymbol{\omega}_n$'s, as well as the motors' speeds $[P_0, P_1, P_2, P_3]_n$'s for all vehicles.

$$\{n : [\mathbf{x}_n, \mathbf{q}_n, r_n, p_n, j_n, \dot{\mathbf{x}}_\mathbf{n}, \boldsymbol{\omega}_n, [P_0, P_1, P_2, P_3]_n]\}. \tag{7}$$

1) Adjacency Matrix of Multi-Robot Systems: In networked multi-robot systems, it is convenient to express the notion of *neighboring robots* (those within a certain radius R) through adjacency matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ A_{ij}. In this type of observation, the value of each drone's key is a dictionary that also includes the drone's corresponding row in $\mathbf{A}$:

$$\{n : \{state : [\dots], neighbors : [A_{n0}, \dots, A_{nN}]\}\}. \tag{8}$$

2) Vision and Rendering: Furthermore, leveraging PyBullet's bindings to TinyRenderer/OpenGL3 and inspired by Bitcraze's AI-deck, `gym-pybullet-drones` observations can include video frames in each drone's perspective (towards the positive direction of the local x-axis) for the RGB ($\mathbf{C}_n \in \mathbb{R}^{64\times48\times4}$), depth, and segmentation ($\mathbf{U}_n, \mathbf{O}_n \in \mathbb{R}^{64\times48}$) views.

$$\{n : \{\dots, rgb : \mathbf{C}_n, dep : \mathbf{U}_n, seg : \mathbf{O}_n\}\}. \tag{9}$$

E. Action Spaces

Advancing a `gym-pybullet-drones` environment by one step requires to pass an action (or control input) to it. For the sake of flexibility, and acknowledging that different robotic applications (e.g., stabilization vs. path planning) require different levels of abstractions, we provide multiple implementations.

1) Propellers' RPMs: The default action space of `gym-pybullet-drones` is a dictionary whose keys are the drone indices $n \in [0..N]$ and the values contain the corresponding 4 motor speeds, in RPMs, for each drone:

$$\{n : [P_0, P_1, P_2, P_3]_n\}. \tag{10}$$

2) Desired Velocity Input: Alternatively, drones can be controlled through a dictionary of desired velocity vectors, in the following format:

$$\{n : [v_x, v_y, v_z, v_M]_n\}, \tag{11}$$

where v_x, v_y, v_z are the components of a unit vector and v_M is the desired velocity's magnitude. In this case, the translation of the input into PWMs and motor speeds is delegated to a PID controller comprising of position and attitude control subroutines [27].

3) Other Control Modes: Developing additional RL and MARL applications will likely require to tweak and customize observation and action spaces. The modular structure of `gym-pybullet-drones` is meant to facilitate this. In Section V, we provide learning examples based on one-dimensional action spaces. The inputs of class `DynAviary` are the desired thrust and torques—from which it derives feasible RPMs using non-negative least squares.

F. Learning Workflows

Having understood how `gym-pybullet-drones`'s dynamics and observations/actions spaces work, using it in an RL workflow only requires a few more steps.

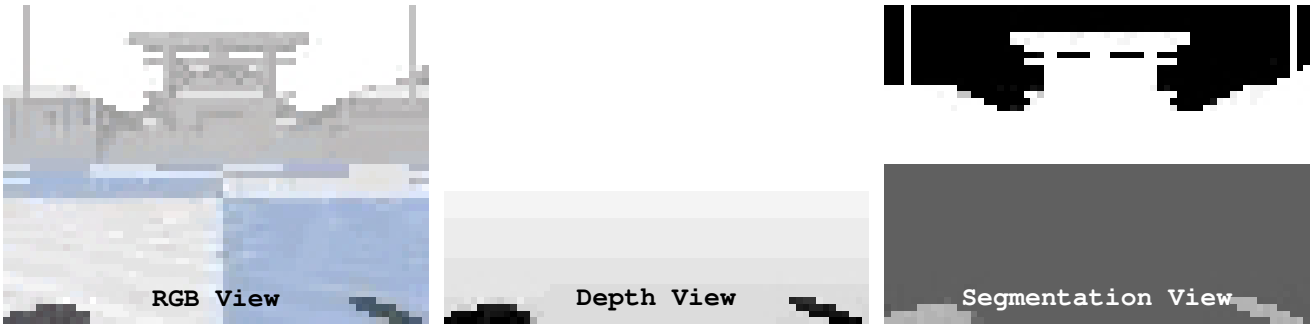

Fig. 5. RGB $\mathbf{C}$, depth $\mathbf{U}$, and segmentation $\mathbf{O}$ `obs` in (7). A Crazyflie 2.x can be given image processing capabilities by the AI-deck.

TABLE II
CPU AND GPU SPEED-UPS AS A FUNCTION OF THE NUMBER VEHICLES, ENVIRONMENTS, AND THE USE OF VISION-BASED OBSERVATIONS

# of Drones	# of Env's	Vision	TinyRenderer‡	OpenGL3‖
1.0	1.0	No	$\mathbf{16.8}\times$	$15.5\times$
1.0	1.0	Yes	$1.3\times$	$\mathbf{10.8}\times$
5.0	1.0	Yes	$0.2\times$	$\mathbf{2.5}\times$
10.0	1.0	No	$2.3\times$	$2.1\times$
80.0	4.0	No	$\mathbf{0.95}\times$	$0.8\times$

‡ 2020 MacBook Pro (CPU: i7-1068NG7)
‖ Lenovo P52 (CPU: i7-8850H; GPU: Quadro P2000)

1) Reward Functions and Episode Termination: Each *step* of an environment should return a reward value (or a dictionary of them, for multiple agents). *Reward* functions are very much task-dependent and one must be implemented. As shown in Section V, it can be as simple as a squared distance. *Gym*'s `done` and `info` return values are optional but can be used, e.g., to implement additional safety requirements [16].

2) Stable Baselines3 Workflow: We provide a complete training workflow for single agent RL based on Stable Baselines3 [11]. This is a collection of RL algorithms—including A2C, DDPG, PPO, SAC, and TD3—implemented in PyTorch. As it supports both MLP and CNN policies, Stable Baselines3 can be used with either kinematics or vision-based observations. In Section V, we show how to run a training example and replay its best performing policy.

3) RLlib Workflow: We also provide an example training workflow for multi-agent RL based on RLlib [12]. RLlib is a library built on top of Ray's API for distributed applications, which includes TensorFlow and PyTorch implementations of many popular RL (e.g., PPO, DDPG, DQN) and MARL (e.g., MADDPG, QMIX) algorithms. In Section V, we show how to run a 2-agent centralized critic training example and replay its best performing policies.

G. ROS2 Wrapper Node

Finally, because of the significance of ROS for the robotics community, we also implemented a minimalist wrapper for `gym-pybullet-drones`'s environments using a ROS2 Python node. This node continuously steps an environment while *(i)* publishing its observations on a topic and *(ii)* reading actions from a separate topic it subscribed to.

IV. COMPUTATIONAL PERFORMANCE

To demonstrate our proposal, we first analyze its computational efficiency. Being able to collect large data sets—through parallel execution and running faster than the wall-clock—is of particular importance for the development of reinforcement learning applications. We chose to adopt *Gym*'s Python API [5], while leveraging Bullet's C++ back end [10], to strike a balance between readability and portability, on one side, and computational performance, on the other.

As we believe that closed-loop performance is the better gauge of a simulation at work, we used a stripped-down version of the PID control [27] example in Figure 6 to generate the data presented in Table II. The script—with no GUI, no debug information, fewer obstacles, and less front end reporting between physics steps—is available in folder: `gym-pybullet-drones/experiments/performance/`.

Unlike simulations that rely on game engines like Unreal Engine 4 and Unity [7], [8], PyBullet has less demanding rendering requirements and can run with either the CPU-based TinyRenderer or OpenGL3 when GPU acceleration is available. We collected the results in Table II using two separate laptop workstations: *(i)* a 2020 MacBook Pro with an Intel i7-1068NG7 CPU and *(ii)* a Lenovo P52 with an Intel i7-8850H CPU and an Nvidia Quadro P2000 GPU.

For a single drone with physics updates at 240Hz, we achieved speed-ups of over $15\times$ the wall-clock. Exploiting parallelism (i.e., multiple vehicles in multiple environments), we could generate $80\times$ the data of the elapsed time. This is slightly slower, but comparable, to Flightmare's dynamics open-loop performance. Although on simpler scenes, a visual observations throughput of $\sim$750kB/s with TinyRenderer is also comparable to Flightmare, and $10\times$ faster when OpenGL3 acceleration is available.

V. EXAMPLES

In practice, we show how one can jointly use `gym-pybullet-drones` with both control approaches and reinforcement learning algorithms. We do so through a set of six examples. All of our source code is available online and it can be installed using the following steps (please refer to the repository for a full list of requirements):

```
$ git clone -b paper \
    git@github.com:utiasDSL/gym-pybullet-drones.git
$ cd gym-pybullet-drones/
$ pip3 install -e .
```

A. Control

The first four examples demonstrate how to command multiple quadcopters using motor speeds or desired velocity

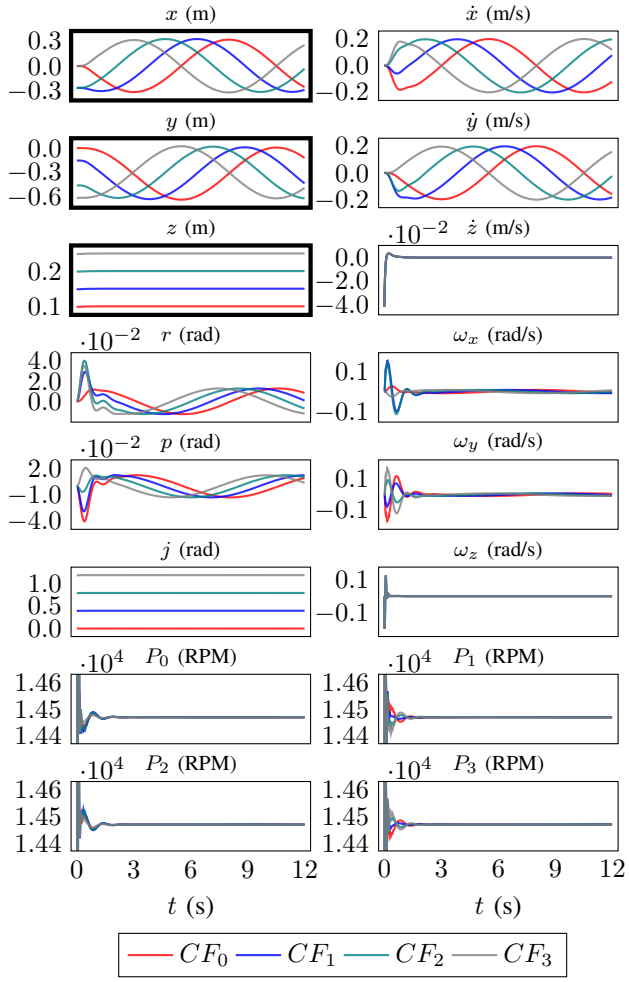

Fig. 6. Positions in x, y, z, linear velocities $\dot{x}$, $\dot{y}$, $\dot{z}$, roll r, pitch p, yaw j, angular velocities $\boldsymbol{\omega}$, and motors' speeds P_0, P_1, P_2, P_3 of four Crazyflies CF_0, CF_1, CF_2, CF_3 tracking a circular trajectory, at different altitudes z's, with different yaws j's, *via* external PID control.

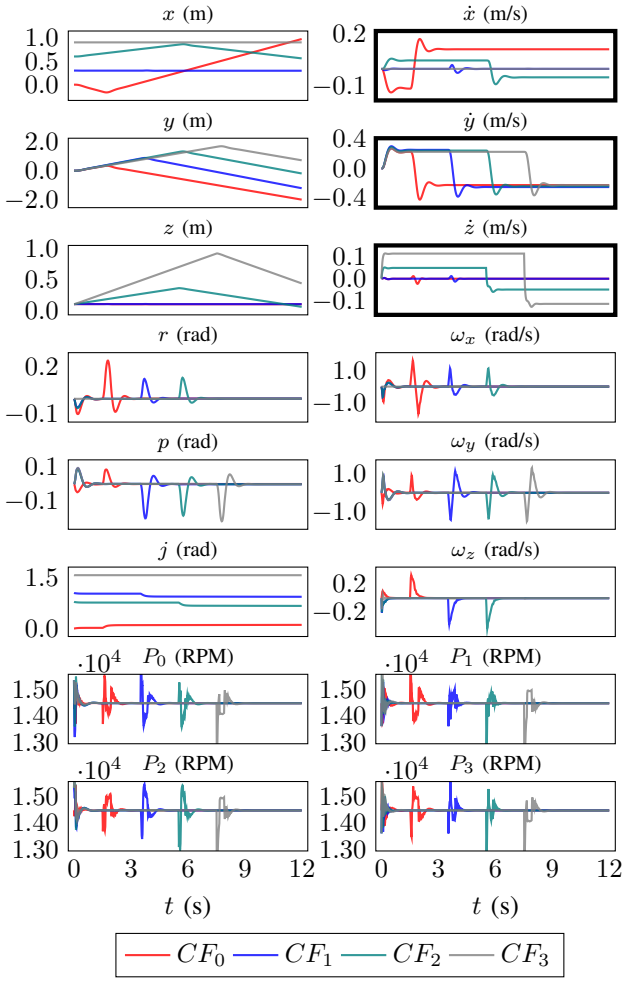

Fig. 7. Positions in x, y, z, linear velocities $\dot{x}$, $\dot{y}$, $\dot{z}$, roll r, pitch p, yaw j, angular velocities $\boldsymbol{\omega}$, and motors' speeds P_0, P_1, P_2, P_3 of four Crazyflies CF_0, CF_1, CF_2, CF_3 tracking step-wise desired velocity inputs *via* PID control embedded within the *Gym* environment.

control inputs as well as two of the aerodynamic effects discussed in Section III: ground effect and downwash.

```
$ cd gym-pybullet-drones/examples/
```

1) Trajectory tracking with PID Control: The first example includes 4 Crazyflies in the $\times$ configuration, using PyBullet's physics update (1), and *external* PID control. The controllers receive the kinematics observations (7) and return commanded motors' speeds (10).

```
$ python3 fly.py --num_drones 4
```

Figure 6 plots position $\mathbf{x}$, velocity $\dot{\mathbf{x}} = [\dot{x}, \dot{y}, \dot{z}]$, roll r, pitch p, yaw j, angular velocity $\boldsymbol{\omega}$, and motors' speeds $[P_0, P_1, P_2, P_3]$ for all vehicles, during a 12 seconds flight along a circular trajectory (the three top-left subplots).

2) Desired Velocity Input: The second example also uses kinematics observations (7) but it is controlled by desired velocity inputs (11). These are targeted by PID controllers *embedded* within the environment.

```
$ python3 velocity.py --duration_sec 12
```

Figure 7 shows the linear velocity response to step-wise changes in the velocity inputs (the three top-right subplots).

3) Ground Effect: The third example compares the take-off of a Crazyflie with and without the ground effect contribution (5), using the coefficient identified in Figure 4.

```
$ python3 groundeffect.py
```

Figure 8 compares positions and velocities along the global z-axis, during the first half-second of simulation time, showing a small but noticeable overshooting and the larger maximum velocity of the drone experiencing the ground effect.

4) Downwash: The last control example subjects two Crazyflies—moving in opposite directions along sinusoidal trajectories in x and different altitudes of 0.5 and 1 meter—to the downwash model in (6).

```
$ python3 downwash.py --gui False
```

Figure 9 compares the simulation results with the experimental data collected to identify the parameters used in (6). Figure 9 shows a very close match in the x and z positions between our simulation and the real-world.

As we would expect, however, our simplified single contribution modeling does not fully capture the impact of downwash on the bottom drone—e.g., in the pitch p and its ramifications on velocity $\dot{x}$.

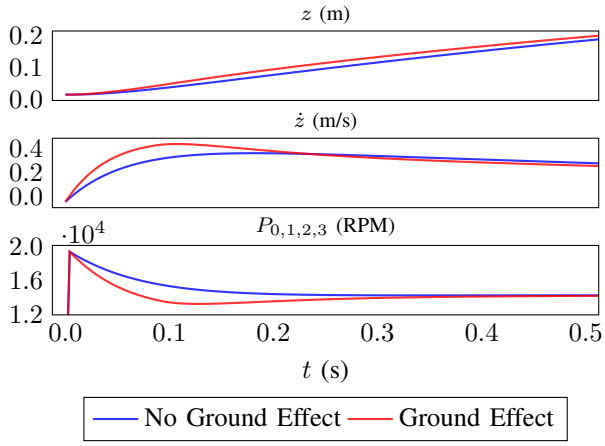

Fig. 8. Position in z, linear velocity $\dot{z}$, and motors' speeds $P_0 = P_1 = P_2 = P_3$ of a simulated Crazyflie taking-off with (red) and without (blue) the modeled ground effect contribution in (5).

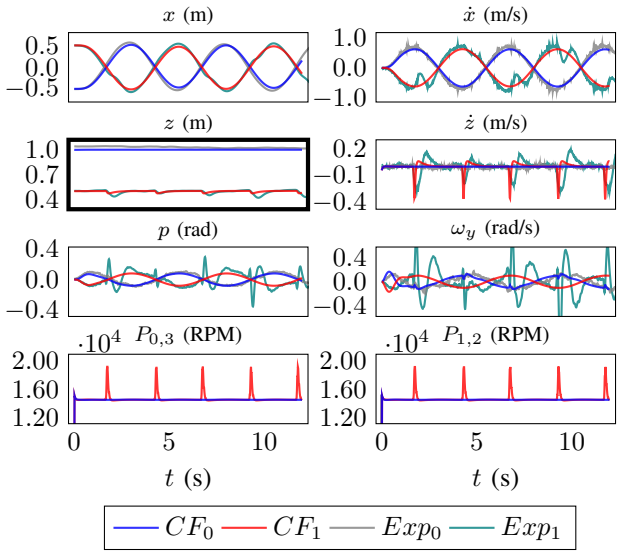

Fig. 9. Positions in x and z, linear velocities $\dot{x}$, $\dot{z}$, pitch p, angular velocity ω_y, and motors' speeds $P_0 = P_3$, $P_1 = P_2$ of two Crazyflies CF_0 (blue), CF_1 (red) subject to the downwash model in (6), compared to the flight logs of a real-world experiment with two drones (grey and teal lines).

B. Reinforcement Learning

The last two examples let one or more quadcopters learn policies to reach a target altitude and hover in place. These are based on *normalized* kinematics observations (7) and a *normalized*, one-dimensional RPMs action space (10).

```
$ cd gym-pybullet-drones/experiments/learning/
```

1) Single Agent Take-off and Hover: For a single agent, the goal is to reach a predetermined altitude and stabilize. The reward function is simply the negation of the squared Euclidean distance from the set point:

$$r = -\|[0, 0, 1] - \mathbf{x}\|_2^2. \tag{12}$$

We use the default implementations of three popular RL algorithms (PPO, A2C, and SAC) provided in Stable Baselines3 [11]. We do not tune any of the hyperparameters. We choose MLP models with ReLU activation and 4 hidden layers with 512, 512, 256, and 128 units, respectively. For PPO, the training workflow can be executed as follows:

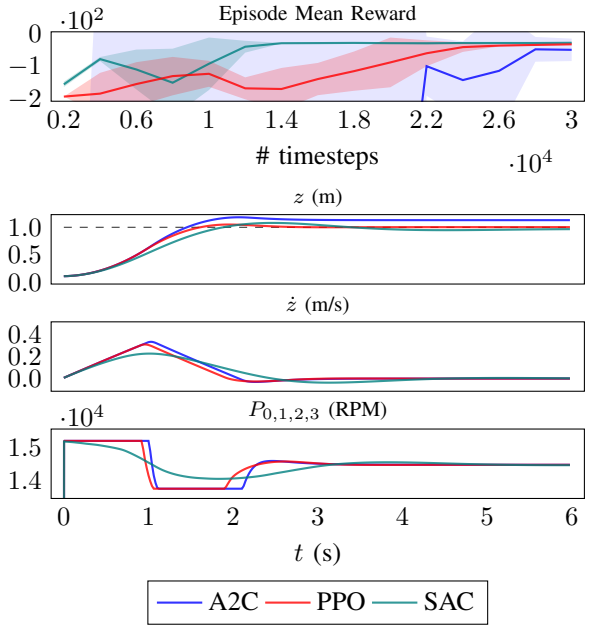

Fig. 10. Algorithm's learning curve (top) and best policy's position in z, linear velocity $\dot{z}$, and motors' speeds $P_0 = P_1 = P_2 = P_3$ (bottom) for three single agent RL implementations from Stable Baselines3—A2C, PPO, and SAC—using the reward function in (12).

```
$ python3 singleagent.py --algo ppo
```

To replay the best trained model, execute the following script:

```
$ python3 test_singleagent.py --exp ./results/save
    -<env>-<algo>-<obs>-<act>-<time_date>
```

Figure 10 compares *(i)* the three algorithms' learning and *(ii)* the trained policies performance. While SAC performs best, all algorithms succeed albeit with very different learning curves. These were not unexpected, as we deliberately omitted any parameter tuning to avoid cherry-picking.

2) Multi-agent Leader-follower: The last example is a MARL problem in which a *leader* agent is trained as in (12) and a *follower* is rewarded by tracking its altitude:

$$r_0 = -\|[0, 0, 0.5] - \mathbf{x}_0\|_2^2, \qquad r_1 = -0.5(z_1 - z_0)^2. \tag{13}$$

The workflow is built on top of RLlib [12] using a central critic with 25 inputs and two action models with 12 inputs, all having two hidden layers of size 256 and tanh activations.

```
$ python3 multiagent.py --act one_d_rpm
```

To replay the best trained model, execute the following script:

```
$ python3 test_multiagent.py --exp ./results/save-<
    env>-2-cc-<obs>-<act>-<time_date>
```

Figure 11 shows a stable training leading to successfully trained policies. The leader presents minor oscillations that are, expectedly, reflected by the follower. The RPMs commanded by these policies, however, appear to be erratic.

VI. Extensions

We developed `gym-pybullet-drones` to provide a compact and comprehensive set of features to kick-start RL in quadcopter control. Yet, we also structured its code base for extensibility. Some of the enhancements in the works include: *(i)* the support for heterogeneous quadcopter teams—this can be achieved by importing multiple URDF

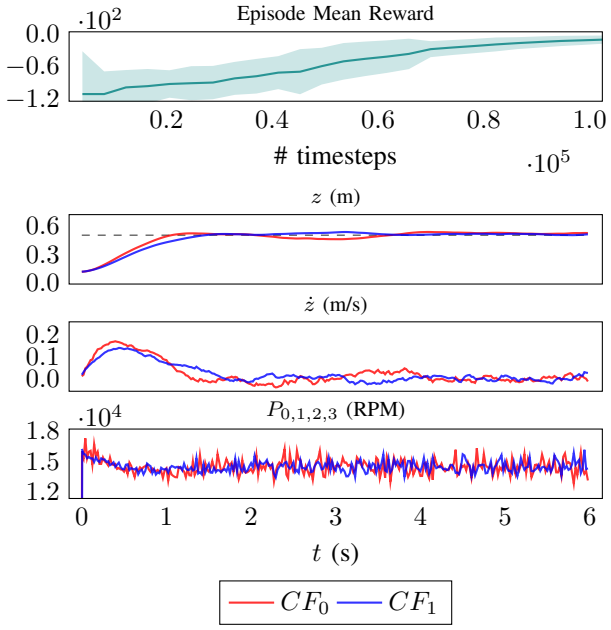

Fig. 11. Learning curve (top) and best policies' positions in z, linear velocities $\dot{z}$, and motors' speeds $P_0 = P_1 = P_2 = P_3$ (bottom) for a 2-agent MARL implementation using RLlib and the reward functions in (13).

files, where the inertial properties are stored; *(ii)* the development of more sophisticated aerodynamic effects—e.g., a downwash model made of multiple components instead of a single contribution applied to the center of mass; *(iii)* the inclusion of symbolic dynamics—e.g., using CasADi to expose an analytical model that could be leveraged by model predictive control approaches; *(iv)* new workflows to support additional MARL frameworks beyond RLlib [12]—e.g., PyMARL; and finally, *(v)* Google Colaboratory support and Jupyter Notebook examples—to facilitate adoption by those with limited access to computing resources.

VII. Conclusions

In this paper, we presented an open-source, OpenAI *Gym*-like [5] multi-quadcopter simulator written in Python on top of the Bullet Physics engine [10]. When compared to similar existing tools, the distinguishing and innovative features of our proposal include *(i)* a more modular and sophisticated physics implementation, *(ii)* vision-based *Gym*'s observations, and *(iii)* a multi-agent reinforcement learning interface. We showed how `gym-pybullet-drones` can be used for low- and high-level control through trajectory tracking and target velocity input examples. We also demonstrated the use of our work in separate workflows for single and multi-agent RL, based on state-of-the-art learning libraries [11], [12]. We believe our work will contribute to bridging the gap between reinforcement learning and control research, helping the community to develop realistic MARL applications for aerial robotics.

Acknowledgments

We acknowledge the support of Mitacs's Elevate Fellowship program and General Dynamics Land Systems-Canada (GDLS-C)'s Innovation Cell. We also thank the Vector Institute for providing access to its computing resources.

References

[1] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. D. Guo, and C. Blundell, "Agent57: Outperforming the Atari human benchmark," in *Proceedings of the 37th International Conference on Machine Learning*, vol. 119. PMLR, 13–18 Jul 2020, pp. 507–517.

[2] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research*, 2013.

[3] B. Recht, "A tour of reinforcement learning: The view from continuous control," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 2, no. 1, pp. 253–279, 2019.

[4] C. K. Liu and D. Negrut, "The role of physics-based simulators in robotics," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, no. 1, 2021.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[6] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *AAAI*, 2018, pp. 3207–3214.

[7] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Proceedings of the 4th Conference on Robot Learning*. PMLR, 2020.

[8] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics*. Springer Int'l Publishing, 2018, pp. 621–635.

[9] G. Silano and L. Iannelli, *CrazyS: A Software-in-the-Loop Simulation Platform for the Crazyflie 2.0 Nano-Quadcopter*. Springer Int'l Publishing, 2020, pp. 81–115.

[10] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2019.

[11] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.

[12] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning*, vol. 80. PMLR, 10–15 Jul 2018, pp. 3053–3062.

[13] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033.

[14] Y. Tassa, S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, and N. Heess, "dm_control: Software and tasks for continuous control," 2020.

[15] M. Chevalier-Boisvert, L. Willems, and S. Pal, "Minimalistic gridworld environment for openai gym," https://github.com/maximecb/gym-minigrid, 2018.

[16] A. Ray, J. Achiam, and D. Amodei, "Benchmarking Safe Exploration in Deep Reinforcement Learning," 2019.

[17] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester, "An empirical investigation of the challenges of real-world reinforcement learning," 2020.

[18] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for uav attitude control," *ACM Transactions on Cyber-Physical Systems*, vol. 3, no. 2, p. 22, 2019.

[19] A. Molchanov, T. Chen, W. Hönig, J. A. Preiss, N. Ayanian, and G. S. Sukhatme, "Sim-to-(multi)-real: Transfer of low-level robust control policies to multiple quadrotors," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2019, pp. 59–66.

[20] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Springer Int'l Publishing, 2016, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625.

[21] S. Krishnan, B. Borojerdian, W. Fu, A. Faust, and V. J. Reddi, "Air learning: An ai research platform for algorithm-hardware benchmarking of autonomous aerial robots," 2019.

[22] C. Luis and J. Le Ny, "Design of a trajectory tracking controller for a nanoquadcopter," Polytechnique Montreal, Tech. Rep., 2016.

[23] J. Förster, "ETH Zurich," Master's thesis, System Identification of the Crazyflie 2.0 Nano Quadrocopter, 2015.

[24] B. Landry, "Planning and control for quadrotor flight through cluttered environments," Master's thesis, MIT, 2015.

[25] C. Powers, D. Mellinger, and V. Kumar, *Quadrotor Kinematics and Dynamics*. Springer Netherlands, 2015, pp. 307–328.

[26] G. Shi, X. Shi, M. O'Connell, R. Yu, K. Azizzadenesheli, A. Anandkumar, Y. Yue, and S. Chung, "Neural lander: Stable drone landing control using learned dynamics," in *International Conference on Robotics and Automation*, 2019, pp. 9784–9790.

[27] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *International Conference on Robotics and Automation*, 2011, pp. 2520–2525.