– Supplementary Materials –

Sim-to-Real Transfer for Low-Level Control of a Quadrotor UAV with Multi-Agent Reinforcement Learning

Beomyeol Yu and Taeyoung Lee

1 Embedded Systems Development

1.1 Hardware Development

To facilitate sim-to-real transfer, we implemented a custom hardware platform that tightly integrates onboard components, including a flight computer, sensors, and actuators, ensuring robust and reliable operation.

An *NVIDIA Jetson TX2* module, running Ubuntu 18.04 and JetPack 4.6, served as the onboard flight computer, responsible for reading sensor measurements, performing all necessary calculations for estimation and control, and sending commands to the actuators. A custom PCB board was designed and manufactured to facilitate the connection between the flight computer, sensors, and actuators while ensuring proper voltage regulation.

For measuring acceleration and angular velocity of the quadrotor, a 9-axis onboard inertial measurement unit (IMU) sensor, a *VectorNav VN100*, operated at 200 Hz with minimal latency. While IMU sensors are capable of providing attitude measurements, magnetic field disturbances within a building can potentially impact the accuracy of yaw direction measurements. Consequently, to precisely measure the quadrotor's position and attitude during indoor flight testing, a Vicon Motion Capture system was utilized, employing six cameras to track reflective markers placed on the quadrotor frame. The captured measurements were processed on an off-board server connected to a network and then transmitted to the onboard computer through Wi-Fi, operating at 100 Hz.

The actuator system consisted of four *T-Motor 700KV* DC brushless electric motors paired with *MS1101* polymer propellers. Each motor required a separate electronic speed controller (ESC), specifically *MikroKopter BL-Ctrl v2* modules, and these ESCs were powered directly from a single 14.8V 4-Cell LiPo battery. During flight, the flight computer sent motor commands to the ESCs as I2C signals, prompting the ESCs to distribute power from the battery to the motors and execute the desired motor commands.

1.2 Software Architecture

We designed a custom flight software for deploying and testing policies trained with the proposed frameworks. For seamless sim-to-real transfer with minimal latency, the onboard Jetson TX2 handles most flight-related computations. The flight code consists of three core modules: sensors, estimators, and RL-based controllers. As represented by the blue shaded box, the majority of the code, including sensor and estimator threads, adopts a multiple-thread structure using the C++ standard thread library. In contrast, the RL-based controller, as depicted by the red shaded box, is implemented in Python, deploying pre-trained PyTorch models. Further, the ground station serves as a communication hub, sending commands (e.g., desired trajectory and motor on/off) and receiving data for real-time monitoring.

Specifically, all sensors operate asynchronously, and communication with them is established through separate threads, namely the VICON and IMU threads. Upon receiving a new asynchronous measurement, each sensor thread places the message into a first-in, first-out (FIFO) buffer, referred to as the sensor message buffer. Then, the Extended Kalman Filter (EKF), operating in a separate estimator thread, fuses sensor measurements to estimate the states $(\hat{x}, \hat{v}, \hat{R}, \hat{\Omega})$. These estimated states are then published to the Python-based RL controller via ROS2 and transmitted to the ground station for real-time data monitoring through a Wi-Fi connection. Next, the low-level control policies are executed independently in a separate module that subscribes to both the states updated by the estimator and the desired commands provided by the user. The controller then generates the desired motor commands $T_{1:4}$ and pushes them into a thread-safe FIFO buffer, called the motor command buffer. Lastly, the motor control thread reads the front-most message from the motor command buffer and adjusts the brushless DC motors at the required speed.

2 Policy Learning Details

2.1 Training Method

Our frameworks are designed to be compatible with any single and multi-RL algorithms. In this work, we employed the TD3 algorithm [1] for SARL and DTDE frameworks and the Multi-Agent TD3 algorithm [2] for the CTDE framework. We trained them for 1.6 million timesteps on a GPU workstation equipped with NVIDIA A100-PCIE-40GB. At each step, the reward signal was normalized to the range $[0, 1]$ for robust convergence. To enhance training speed and stability, both state and action were also normalized to the range $[-1, 1]$ during training and evaluation. Additionally, all states were randomly drawn from pre-defined distributions at the beginning of an episode to encourage diverse exploration, e.g., the quadrotor was randomly placed within a $3\text{m} \times 3\text{m} \times 3\text{m}$ space. To prevent overfitting, we adopted the SGDR learning rate scheduler [3] and applied linear decay to the exploration noise while updating the policies. This means that the amount of randomness in the policy gradually decreases as the training progresses. Notably, unlike existing approaches that often rely on auxiliary techniques or pre-training, our agents successfully learned complex behaviors through model-free learning.

2.2 Hyperparameters

Table 1: Hyperparameters.

Parameter	Value
Optimizer	AdamW
Learning rate	$1 \times 10^{-4} \to 1 \times 10^{-6}$
Discount factor, γ	0.99
Replay buffer size	10^6
Batch size	256
Maximum global norm	100
Exploration noise	$0.3 \to 0.05$
Target policy noise	0.2
Policy noise clip	0.5
Target update interval	3
Target smoothing coefficient	0.005

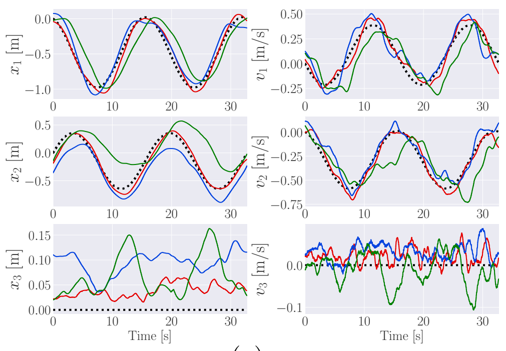
To prioritize accurate trajectory tracking, the reward coefficients for position and heading were tuned to higher values, specifically $k_v = 7.0$ and $k_{b_1} = 2.0$, respectively. Also, the penalizing terms were carefully adjusted to ensure smooth and stable control, that is, $k_v = k_\Omega = k_{\omega_{12}} = 0.25$, $k_{b_3} = 3.5$, and $k_{\Omega_3} = 0.2$. These were crucial, as too large penalties were found to over-prioritize stabilization and thus hinder the quadrotor's movement. To eliminate the steady-state errors, k_{I_x} and $k_{I_{b_1}}$ were set to 0.2 and 0.1, with $\alpha = 0.1$ and $\beta = 0.01$, respectively. Table 1 summarizes the hyperparameters utilized throughout training.

3 Hovering and circular trajectory tracking

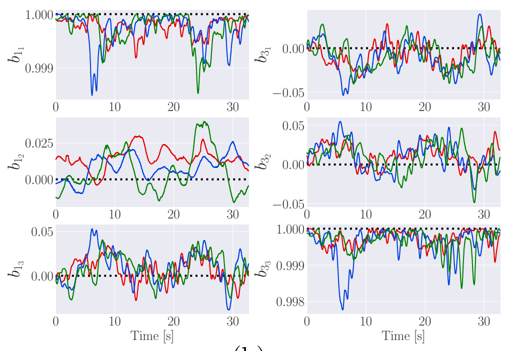
To validate the sim-to-real adaptation capabilities of our RL frameworks, we performed real-world flight experiments in which the transferred agents tracked desired positions x_d in real-time while maintaining the desired directions $b_{1_d} = [1, 0, 0]^T$ and $b_{3_d} = [0, 0, 1]^T$. We compare the performance of each framework with a traditional geometric control during hovering and circular trajectory tracking tasks, under near-zero initial yawing error. Table 2 summarizes a comparison of their real-time flight performance, measured in terms of average position errors e_x in centimeters and yawing errors e_{b_1} in degrees. Additionally, to illustrate the performance of our proposed frameworks, we present the flight trajectories on the circular trajectory tracking task. The trajectories of CTDE, DTDE, and SARL are depicted in red, blue, and green lines, respectively. To compare the tracking performance, we visualize their position and velocity trajectories in Figure 1–(a), and first and third directions, b_1 and b_3, results in Figure 1–(b). While SARL maintained the desired yaw command b_{1_d}, its tightly coupled nature led to significant fluctuations in translational control. Unlike SARL, DTDE and CTDE effectively eliminate b_1 error while achieving smaller average e_x and e_v by decoupling yaw control. Lastly, Figure 1–(c) depicts the motor thrust $T_{1:4}$ of each framework, which shows that the policy regularization technique achieved smooth control, generating practical control signals.

Table 2: Real-world flight performance comparison of each RL framework and a traditional geometric control for hovering and circular trajectory tracking when $e_{b_1}(0) \approx 0$, measured by average position e_x and heading e_{b_1} errors.

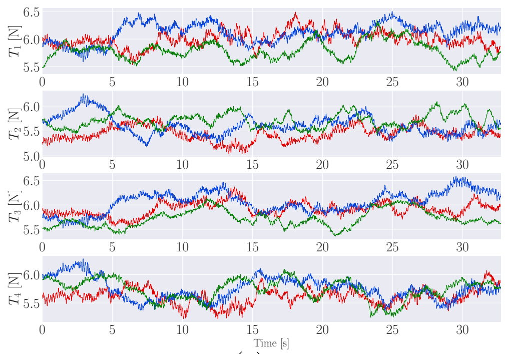
Policy	Hovering		Circular Trajectory	
	e_x[cm]	e_{b_1}[°]	e_x[cm]	e_{b_1}[°]
SARL	(7.28, 9.79, 7.08)	0.82	(12.4, 20.5, 6.82)	0.67
DTDE	(7.10, 2.71, 7.37)	0.30	(10.1, 14.8, 9.49)	0.53
CTDE	(3.16, 2.90, 0.53)	0.33	(5.10, 6.31, 3.86)	0.85
GEOM	(3.19, 2.50, 0.71)	0.98	(6.16, 5.37, 1.38)	0.35



(a)



(b)



(c)

Figure 1: Real-world flight performance comparison of each RL framework and a traditional geometric control for circular trajectory tracking when $e_{b_1}(0) \approx 0$, measured by average position e_x and heading e_{b_1} errors.

References

[1] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning.* PMLR, 2018, pp. 1587–1596.

[2] J. Ackermann, V. Gabler, T. Osa, and M. Sugiyama, "Reducing overestimation bias in multi-agent domains using double centralized critics," *arXiv preprint arXiv:1910.01465*, 2019.

[3] I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.