– Supplementary Materials –

Modular Reinforcement Learning for a Quadrotor UAV with Decoupled Yaw Control

Paper: `TBD`
Video: `https://youtu.be/-NQ6oRsdWgI`

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 for 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 200 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.

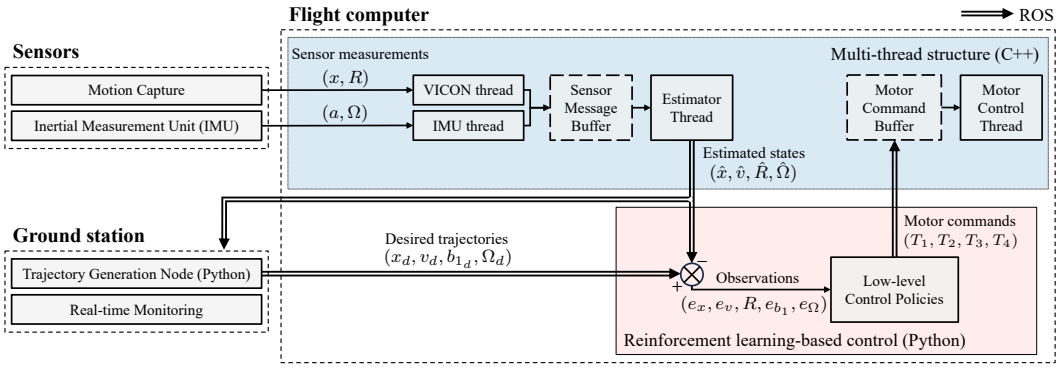

Figure 1: Modular flight software architecture for policy evaluation in real-world environments. Most of the flight code, including sensor, estimator, and motor control threads, is designed in C++ using a multi-threaded architecture (blue shaded box), while the RL-based controller is implemented in Python to deploy *PyTorch* models (red shaded box), interacting with C++ code via ROS. The motion capture system and onboard IMU sensors measure the quadrotor's acceleration and angular velocity (a, Ω), and its position and attitude (x, R), respectively, and then each raw measurement is fed to the VICON and IMU threads. Lastly, the ground station communicates with the quadrotor, sending commands and receiving data.

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 frontmost 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 NMP and DMP frameworks and the Multi-Agent TD3 algorithm [2] for the CMP framework. We trained them for 1.5 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 1m^3 cube 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. Table 1 summarizes the hyperparameters utilized throughout training.

Table 1: Hyperparameters.

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

Next, to prioritize accurate trajectory tracking, the reward coefficients for position and heading were tuned to higher values, specifically $k_v = k_{b_1} = 6.0$. Also, the penalizing terms were carefully adjusted to ensure smooth and stable control, that is, $k_v = 0.4$, $k_\Omega = k_{\omega_{12}} = 0.6$, and $k_{\Omega_3} = 0.1$, respectively. These penalizing terms 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 0.1 and 0.1, with $\alpha = 0.1$ and $\beta = 0.05$, respectively. Next, for smooth control, the regularization weights were chosen as $\lambda_T = 0.4$, $\lambda_S = 0.3$, and $\lambda_M = 0.6$, respectively.

2.2 Domain Randomization

While simulators provide theoretically unlimited data, transferring trained policies from simulation to real-world environments encounters challenges due to sim-to-real mismatch. This mismatch often arises from inconsistencies in physical parameters and inaccuracies in physical modeling. To address this, *domain randomization* techniques have emerged as a simple yet promising solution. This method involves randomizing various simulation properties (e.g., mass) during training, enabling agents to adapt and generalize across diverse conditions. In our implementation, the simulator's physical parameters are uniformly sampled from a range of $\pm 10\%$ around the nominal values listed in Table 2 at the beginning of each episode.

Table 2: Quadrotor nominal parameters.

Parameter	Nominal Value
Mass, m	2.15 kg
Arm length, d	0.23 m
Moment of inertia, J	(0.022, 0.022, 0.035) kgm^2
Torque-to-thrust coefficient, $c_{\tau f}$	0.0135
Thrust-to-weight coefficients, c_{tw}	2.2

3 Figure-eight Lissajous trajectory commands

To evaluate the sim-to-real adaptation capabilities, we compare the control performance of each RL framework. The position tracking performance was evaluated using a figure-eight Lissajous trajectory defined by $x_d(t) = [e'(t)\sin(w_1 t)) + x_1(0),\ 1.5\, e'(t)(\cos(w_2 t) - 1) + x_2(0),\ x_3(0)]^T$ where $w_1 = 4\pi/T_d$ and $w_2 = 2\pi/T_d$ with a desired period of the cycle $T_d = 9$ seconds. Here, to mitigate aggressive movements at the beginning of the trajectory, we designed an exponentially decaying term $e'(t) = 1 - e^{-\epsilon t}$, with smoothing factor $\epsilon = -\ln(0.01)/T_d$. Additionally, yawing tracking performance was assessed with $b_{1_d} = [\cos(e'(t)\omega_d t),\ \sin(e'(t)\omega_d t,\ 0]^T$ where ω_d is a desired yaw rate.

4 Straight line trajectory tracking

We further performed real-world flight experiments along a straight line trajectory. The desired trajectory is a straight line specified by $x_d(t) = [x_1(0) + 0.2t, x_2(0), x_3(0)]$ with the desired directions $b_{1_d} = [1, 0, 0]^T$ and $b_{3_d} = [0, 0, 1]^T$. For a comparison of tracking performance, we visualize their position and velocity trajectories in Figure 2–(a), and first and third directions, b_1 and b_3, results in Figure 2–(b), when the initial yaw error is $e_{b_1}(0) = 100°$. While NMP managed to maintain the desired yaw command b_{1_d}, it exhibits significant performance degradation in the position tracking. In contrast, DMP and CMP effectively eliminate yawing errors while achieving smaller averages e_x and e_v by decoupling yaw control. Also,

Figure 2–(c) illustrates the motor thrust of each framework, showing that the policy regularization technique achieved smooth control, generating practical control signals and avoiding undesirable chattering, a crucial contribution to robust quadrotor performance.

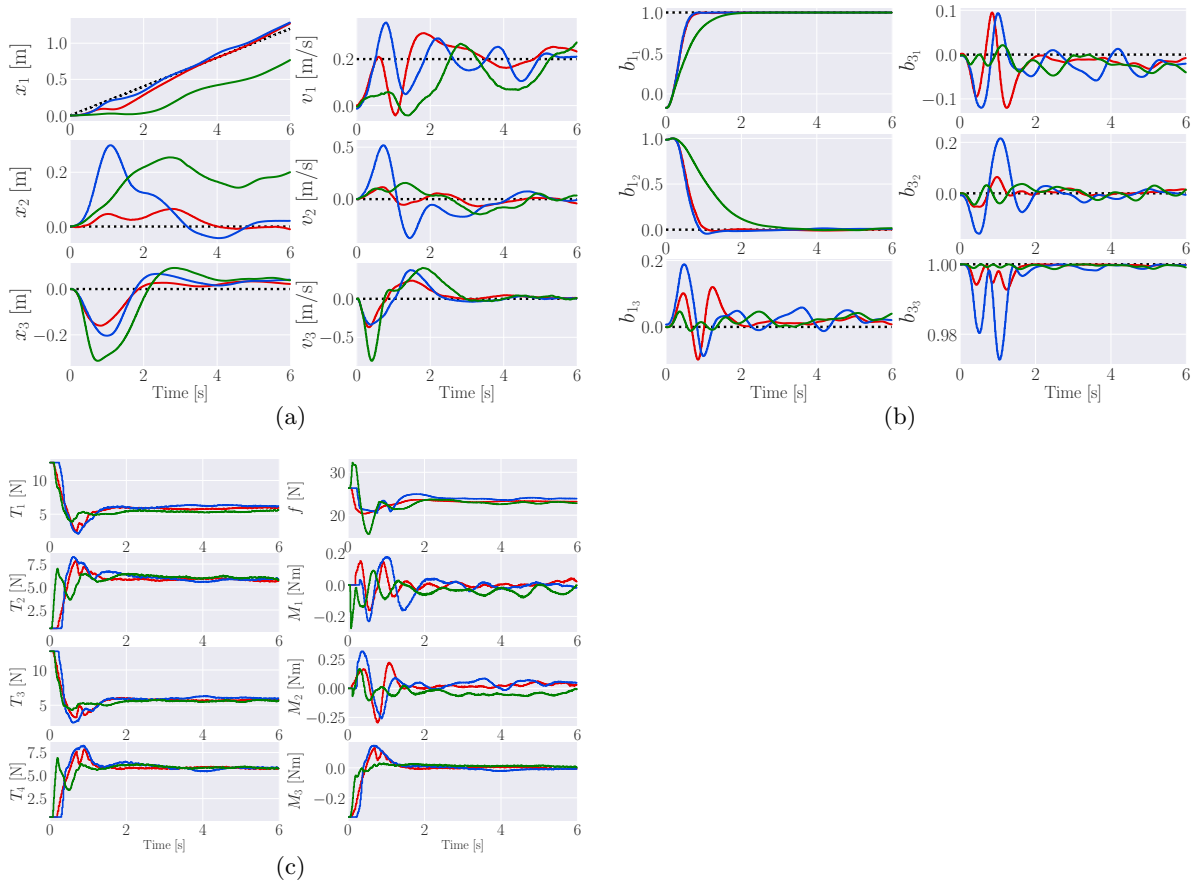

Figure 2: Real-world flight trajectories of NMP (green), DMP (blue), and CMP (red) when $e_{b_1}(0) = 100°$, corresponding to the results in Table 3. (left) position $x \in \mathbb{R}^3$ and velocity tracking $v \in \mathbb{R}^3$. (middle) the first $b_1 \in \mathbb{R}^3$ and third $b_3 \in \mathbb{R}^3$ directions. (right) corresponding motor thrusts $T_{1:4}$, total force f, and moment M, generated by each agent.

Next, Table 3 provides a comparison of their flight performance, measured in terms of average position errors e_x in centimeters and yawing errors e_{b_1} in degrees, when the initial heading error is varied. Both of the proposed modular schemes exhibit consistent performance comparable to the numerical simulation. However, the tracking errors of NMP increase as the initial yaw error, leading to instability when the yaw error is 125°. This illustrates the overall challenges in zero-shot sim-to-real transfer and the advantages of the proposed modular schemes in overcoming sim-to-real gaps.

Table 3: Real-world flight performance comparison of each RL framework and a traditional geometric control, named GEOM, for tracking a straight line trajectory of $x_d(t) = [x_1(0) + 0.2t, x_2(0), x_3(0)]^T$ and $v_d(t) = [0.2, 0, 0]^T$, while converging to $b_{1_d} = [1, 0, 0]^T$ under varying initial yaw errors. Average position $\bar{e}_x$ (in centimeters) and heading $\bar{e}_{b_1}$ (in degrees) errors quantify their performance.

Policy	$e_{b_1}(0) = 25°$		$e_{b_1}(0) = 50°$		$e_{b_1}(0) = 75°$		$e_{b_1}(0) = 100°$		$e_{b_1}(0) = 125°$	
	$\bar{e}_x$	$\bar{e}_{b_1}$	$\bar{e}_x$	$\bar{e}_{b_1}$	$\bar{e}_x$	$\bar{e}_{b_1}$	$\bar{e}_x$	$\bar{e}_{b_1}$	$\bar{e}_x$	$\bar{e}_{b_1}$
NMP	10.27	3.62	13.78	7.94	15.77	10.24	19.92	14.40	crashed	
DMP	5.26	1.15	5.56	2.80	4.66	5.58	5.83	7.47	9.04	9.63
CMP	3.92	1.40	3.19	3.27	3.81	4.91	4.05	6.68	6.74	11.18
GEOM	1.93	2.47	1.92	5.49	2.59	8.92	3.39	12.33	3.96	17.57

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.