Reinforcement Learning for Quadcopter Control

Daniel Bin Schmid, Matej Straka
School of Computation, Information, Technology
Technical University of Munich
{danielbin.schmid,matej.straka}@tum.de

***Abstract*—In this work, we experiment with reinforcement learning (RL) algorithms. In particular, we experiment with a trajectory following RL policy from drone racing. To guide the control algorithms and train the RL policy, we randomly generate feasible time-continuous trajectories utilizing minimum-snap polynomial trajectory generation. This letter documents the implementation details and our findings.**

***Index Terms*—Unmanned Aerial Vehicles, Reinforcement Learning, Trajectory Planning**

I. Introduction

Small drones, or also unmanned aerial vehicles (UAVs), are predicted to have wide socio-economic impact once the legal roadblock is lifted [9]. They can be used in a wide array of applications such as for security and surveillance [12], aiding in rescue operations [1], or delivery and logistic [11]. In context of an university project, we work hands-on with one of such small UAVs, namely a Crazyflie. As part of this endeavor, we implement control mechanisms for the quadcopter, which we document in this letter. In more particular, we experiment with reinforcement learning (RL) based control approaches from drone racing [15] by implementing it using an open-source simulator [14] that connects pybullet physics simulation [6] with Gymnasium [18] and stable-baselines [17], and Bitcraze's Crazyflie firmware [2].

II. Pybullet Simulation

The simulation environment `gym-pybullet-drones` [14] significantly eased experimentation because it sped up simulation speed from $2\times$ in Webots (which we used prior to using `gym-pybullet-drones`) to $20\times$. We extended the simulation by implementing additional features. In particular, attitude-level control in the simulation was implemented since it allows for designing the RL agent to output attitude commands instead of low-level motor commands, which is shown to improve RL based control policies [10]. Furthermore, minimum-snap polynomial trajectory generation from section III written in C++ is integrated by implementing an interface to python via `pybind11`. The code is made available on github[1].

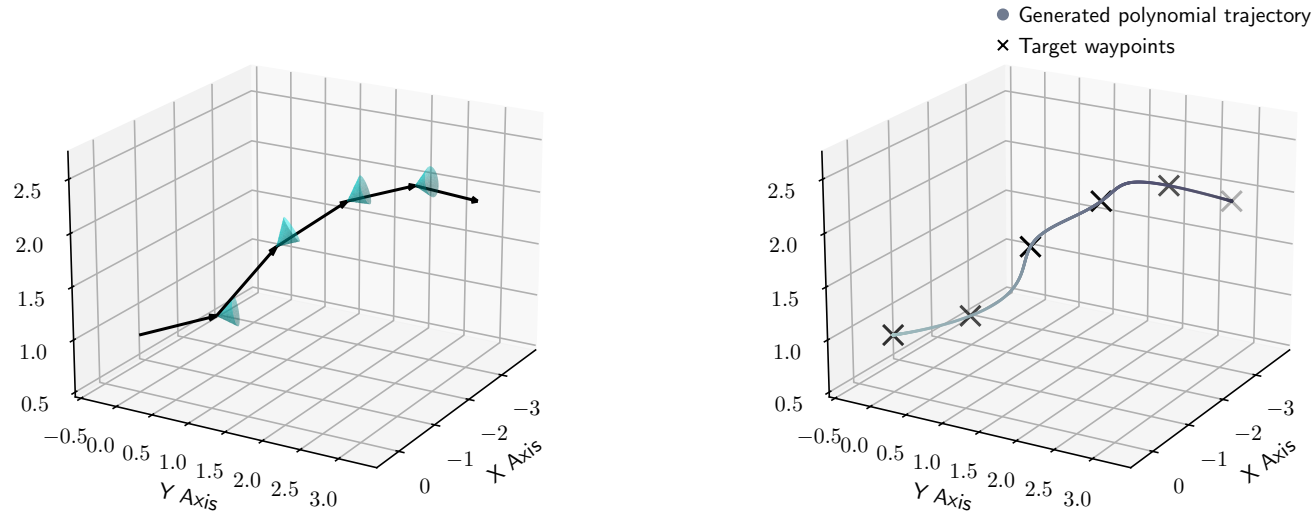
III. Trajectory Generation

To train a generalizing RL policy (see **??**), we generate randomized trajectories during training. This allows the agent to follow arbitrary trajectories during testing, instead of overfitting to a fixed set of trajectories. For this purpose, we adopt the minimum snap/ jerk trajectory generation algorithm (for more details see appendix A1) and feed it with random target/ control waypoints q and uniform timestamps T. Algorithm 1 illustrates our algorithm in pseudocode, where GenTraj$(q_{init}, \vec{v}_{init}, n_{ctrl}, d_{ctrl}, \sigma^2_{deg}, t_\Delta)$ is the main procedure which generates a random trajectory and SampleDirection(σ^2_{deg}) samples a random direction from a gaussian distribution with a standard deviation of σ^2_{deg}, measured in degrees. The algorithm takes an initial position q_{init} and an initial direction $\vec{v}_{init}$. In each iteration, a random direction is sampled by sampling a zenithal and an azimuthal angle from a gaussian distribution with mean 0 and standard deviation σ^2_{deg}, and an uniform distribution in $[0, 2\pi]$, respectively. The azimuthal angle measures the rotation around the vertical (or up) axis, and the zenithal angle measures the

Algorithm 1 Random Polynomial Trajectory Generation

procedure SampleDirection($\sigma^2_{deg} \in [0, 360]$)
  $\sigma^2_{rad} \leftarrow$ DegreesToRadians(σ^2_{deg})
  $\theta_{rad} \leftarrow \mathcal{N}(0, \sigma^2_{rad})$ ▷ Sample zenithal angle
  $\phi_{rad} \leftarrow \mathcal{U}[0, 2\pi]$ ▷ Sample azimuthal angle
  $x \leftarrow \sin(\theta_{rad}) \cdot \cos(\phi_{rad})$
  $y \leftarrow \sin(\theta_{rad}) \cdot \sin(\phi_{rad})$
  $z \leftarrow \cos(\theta_{rad})$
  return $(x, y, z)^T$
end procedure
procedure GenTraj($q_{init} \in \mathbb{R}^3, \vec{v}_{init} \in \mathbb{R}^3, n_{ctrl} \in \mathbb{N}_1, d_{ctrl} > 0, \sigma^2_{deg} \in [0, 360], t_\Delta > 0$)
  $L \leftarrow (q_{init})$ ▷ Sequence of control points
  $T \leftarrow (0)$ ▷ Sequence of timestamps
  $q_{cur}, \vec{v}_{cur} \leftarrow q_{init}, \vec{v}_{init}$
  for $i \in \{1, \dots, n_{ctrl}\}$ **do**
    $M_{rot} \leftarrow$ RotMatFromUpTo(v_{cur})
    $\vec{v} \leftarrow M_{rot} \cdot$ SampleDirection(σ^2_{deg})
    $q \leftarrow q_{cur} + d_{ctrl} \cdot \frac{\vec{v}}{\|\vec{v}\|}$
    $L \leftarrow$ AppendToSequence(L, q)
    $T \leftarrow$ AppendToSequence$(T, i \cdot t_\Delta)$
    $q_{cur}, \vec{v}_{cur} \leftarrow q, \vec{v}$
  end for
  $\sigma(t) \leftarrow$ MinimumSnapPolGen(L, T) ▷ Or with jerk
  return $\sigma(t)$
end procedure

[1]https://github.com/danielbinschmid/RL-pybullets-cf



(a) Random control point generation.

(b) Minimum snap trajectory generation.

Fig. 1: Random smooth trajectory generation using random control point sampling and minimum snap trajectory generation. (a) shows the random control point sampling mechanism where the blue cones visualise the standard deviation for sampling from a gaussian distribution. (b) shows the smooth trajectory generated using [20].

deviation from the vertical axis. The sampled direction then is rotated to align with the current direction $\vec{v}_{cur}$ instead of the up vector, normalised, scaled with d_{ctrl}, and added to the current position q_{cur}. We work with an uniform distance between control points d_{ctrl}, and a uniform time difference t_{Δ}. With this procedure, n_{ctrl} control points are sampled, which are then fed into the minimum snap/ jerk optimization algorithm. For an example of a randomly generated output, see fig. 1. To integrate the random minimum snap trajectory generation into the `gym-pybullet-drones` simulation, we use an open-source C++ implementation by [20][2] and integrate it using `pybind11`.

IV. Waypoint Follower

As we started working on a RL controller, we initially focused on developing a controller able to minimize the drone position relative to a target position using the PPO algorithm. After some experiments, the best model was trained using directly motor commands for controlling the drone. Observation space contained relative distance error, orientation, linear and angular velocity, linear acceleration and the past 4 motor commands. The controller was then trained by optimizing for the following reward:

$$r(\boldsymbol{p}, \boldsymbol{t}) = max(0, 2 - ||\boldsymbol{p} - \boldsymbol{t}||^4) + \mathbf{100}_{\{\|p-t\| \leq r\}}, \qquad (1)$$

where $\boldsymbol{p}, \boldsymbol{t} \in \mathbb{R}^3$ are the drone position and the target position, respectively, and r is a radius where we accept that the drone reached a given target position. Figure 2 shows our obtained policy.

[2]https://github.com/ZJU-FAST-Lab/large_scale_traj_optimizer

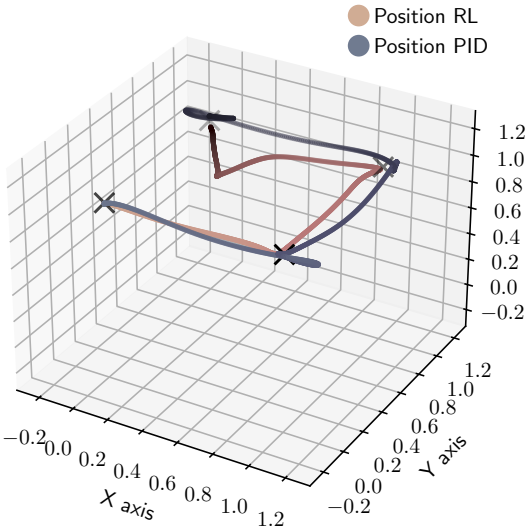


Fig. 2: Simple position controller RL agent on a 4-waypoint following task, compared against a position PID controller.

V. Trajectory Follower

In the area of drone racing, Penicka et al. [15] tackle the problem of trajectory following by introducing (among other contributions) a more sophisticated reward shaping and a richer observation space. Our implementation of reward shaping closely follows the approach outlined in [15]. Similarly, we adopt a similar observation space. In the following, we present our modified approach:

1) Observation space: Our observations $v \in \mathbb{R}^{13+3n}$ consist of quadrotor position, roll, pitch, yaw, linear velocity, angular velocity, projection from the drone to the closest

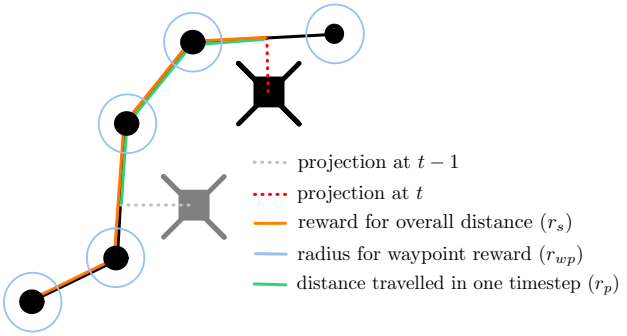


Fig. 3: Illustration of different components of the reward signal. The grey quadrotor represents the agent at timestep $t-1$, the black quadrotor represents the agent at timestep t.

point on its trajectory and n vectors pointing from the drone to subsequent n waypoints. Allowing the agent to perceive its projection to the closest point on a trajectory allows it in combination with a proper reward to learn to follow trajectories more closely. Adding future waypoints, on the other hand, allows it to prepare for the next point after reaching the immediate one.

2) Action space: There are multiple possibilities for an action space. One such candidate is to output 4 scalars (RPMs), which are direct commands to motors. Kaufman et al. [10] have shown that outputting motor level commands directly is inferior to outputting attitude commands. These consist of desired changes of bodyrates (yaw, pitch, roll) and a change in a single constant RPM value that is sent to all four motors.

3) Reward shaping: The reward that is provided as a training signal to an optimization algorithm composes of multiple terms and can be expressed as:

$$r(t) = k_p r_p(t) + k_s r_s(t) + k_{wp} r_{wp} + r_T, \tag{2}$$

where $r_p(t)$ is a difference in travelled distance over trajectory between current and previous timestep, $r_s(t)$ is a total travelled distance over trajectory, r_{wp} is a reward for entering a certain radius of a waypoint (and is given only once per waypoint), r_T is a negative reward for crashing and k_p, k_s, k_{wp} are hyperparameters defining importance of these terms. For full derivation of these terms we refer to [16]. Instead, as a supplement, we provide a fig. 3 illustrating these terms. The reward from eq. (2) produces a learning behavior by itself, but the resulting agent has tendencies to search for shortcuts in a trajectory, since it only maximizes for progressing over it, not for faithfully following it. For this reason, authors [15] introduce a scaling coefficient $\alpha = \alpha_d \alpha_{v_{min}} \alpha_{v_{max}}$ that constitutes of three multiplicands. This coefficient ensures that the agent operates within a given velocity range $[\alpha_{v_{min}}, \alpha_{v_{max}}]$ and also follows trajectory within a desired distance $\leq \alpha_d$. Again, we refer to [15] for more details. The final expression for reward signal is

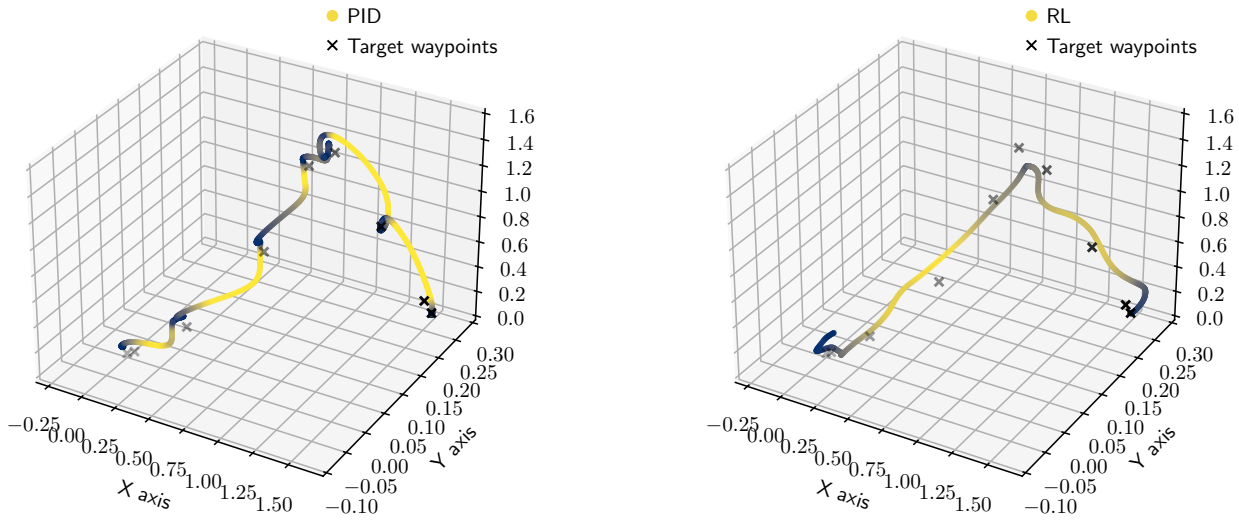
$$r(t) = \alpha k_p r_p(t) + \alpha k_s r_s(t) + k_{wp} r_{wp} + r_T. \tag{3}$$

4) Training: We train the model with random trajectories generated using GENTRAJ from algorithm 1 with $\sigma^2_{deg} = 50$, $q_{init} = (0, 0, 1)^T$, $d_{ctrl} = 1.3$, and $n_{ctrl} = 10$. Trajectories are generated in open space, meaning that there are no obstacles in the way. We use the PPO algorithm, with a learning rate of 0.0003 and train for 2.5M timesteps.

5) Experiments: One of the main difficulties of the RL approach is to properly adjust hyperparameters, in our case k_p, k_s, k_{wp} as well as velocity ranges and desired distance to a trajectory. The problem is that the rewards associated with these coefficients are complementary and setting them improperly leads to an undesired behavior. For example, setting a large k_s makes the agent focus mainly on getting to the final waypoint as quickly possible, which results in large shortcuts. We ran a grid search over possible hyperparameters in cloud, obtaining 60 trained models and analyzed results quantitatively from plots. However, hyperparameters that we search for directly influence scale of the reward and the performance needs to be evaluated qualitatively, by looking at the behavior of each trained agent. We found that the k_p (change in travelled distance) is the most important value to set properly for overall performance. The reward for overall travelled distance showed to be not useful. We want our agent to be invariant w.r.t. the stage of trajectory it is in. In other words, we want it to behave the same at the beginning and the end of any trajectory. We think that the reward term r_s is useful for drone racing, where we want to incentivize an agent to complete its trajectory quickly. In our case, setting k_s to a larger value causes the drone to cut trajectories in an undesired fashion. We also found that not limiting maximum velocity leads the agent to local optima where it randomly goes into one direction very rapidly in order to obtain rewards quickly before it crashes. This might be resolved by increasing number of parallel agents. Our final agent has parameter values $k_p = 5, k_s = 0.05, k_{wp} = 8$, with maximum desired distance from trajectory set to zero, and a desired velocity range from 0.2 to 1.5 m/s.

VI. EVALUATION

1) Test Set Generation: To evaluate our trajectory following policy for a landing task, we generate a test set of 200 polynomial trajectories with algorithm 1. The design motivation behind this test set is to benchmark performance during a landing operation. We choose $n_{ctrl} = 3$ and $d_{ctrl} = 1.3m$, resulting in $\approx 5.2m$ landing trajectories. We fix the landing position to be $q_{land} := (0, 0, 0)^T$. First, we set $q_{init} := q_{land}$. To ensure that q_{land} is the end position instead of the start position, we reverse the sampled control points L. In more particular, after sampling the control points $L = (q_{init}, q_1, \ldots, q_{n_{ctrl}})$ in algorithm 1, we reverse the sequence to $L^{-1} = (q_{n_{ctrl}}, \ldots, q_1, q_{init})$, and use L^{-1} instead of L, i.e. we call MINIMUMSNAPPOLGEN(L^{-1}, T). To simulate random angles during landing, we choose $\vec{v}_{init}$ as an arbitrary unit vector with z-direction > 0. With this procedure, we randomly generated 200 trajectories.



(a) PID. (b) RL.

Fig. 4: Trajectories are color-coded according to velocity. Blue corresponds to slow, whereas yellow corresponds to fast.

Test Set	Model	Success	Avg Dev (m)	Max Dev (m)	Time (s)
Landing	PID-9	1.0	0.059	0.166	7.98
Landing	RL-10	1.0	0.074	0.155	7.88
Landing	PID-avg	0.89	0.069	0.212	8.22
Landing	RL-avg	1.0	0.092	0.222	**7.22**
Long	PID-23	1.0	0.089	0.228	27.02
Long	RL-26	0.99	0.09	0.23	**22.32**
Long	PID-avg	0.76	0.1	0.292	26.75
Long	RL-avg	0.99	0.126	0.338	**18.37**

TABLE I: Comparison of our reinforcement learning (RL) policy against a position PID controller. Averages and comparable runs are shown.

2) Metrics: Four metrics are computed, the *success rate* [Success], the *mean deviation* from the reference trajectory in meters [Mean Dev (m)], the *maximum deviation* from the reference trajectory in meters [Max Dev (m)], and the *completion time* until successful landing in seconds [Time (s)]. We define [Success] as the state where the drone reached the landing position q_{land} within a distance of $0.2m$ and a low velocity $\leq 5/3\ m/s$ (< 0.05 in the simulation). [Time (s)] is defined as the time it took to reach q_{land} from $q_{n_{ctrl}}$. [Mean Dev (m)] and [Max Dev (m)] are computed by averaging and taking the maximum of the minimum distances of every position visited by the drone to the reference trajectory $\sigma_{land}(t)$, respectively. To compute the distance of a point to $\sigma_{land}(t)$, we apply discretization with 10^4 points and take the smallest distance to the discretized points.

3) Results: To evaluate our results, we compare a position PID controller baseline against our learned policy on the landing task with short trajectories $\approx 5.2m$ and on long trajectories $\approx 27.3m$ without landing. Figure 4a shows the trajectory followed by the PID controller. Note that an unnatural stop-and-go behavior can be observed, which can be mainly attributed to poor usage of the PID controller. In more particular, the PID controller is only given the next waypoint once it reached one target. Nevertheless, by showing that our learned policy improves upon this baseline, we validate our learned policy. In fig. 4b, the smooth trajectory followed by our learned RL policy can be seen. Interestingly, it learned to move slower if the next waypoints in the waypoint buffer are more close, which suggests that the drone learned to correlate velocity with the distance to the next waypoints. Table II shows the benchmarks of this experiment. Overall, the RL policy shows faster completion times and on-par deviation from the reference path compared to the position PID. Note that the PID controller did not achieve 100% success rate for every discretization level due to mis-configured discretization levels. See appendix B for a more detailed description.

VII. Discussion

In this work, we successfully implemented an influential paper from drone racing [15] by building upon the `gym-pybullet-drones` simulation environment. Our policy produces smooth and natural control, as seen in fig. 4b. There are still multiple areas to improve. For example, there is still a room for improvement for making our policy follow the trajectory even more closely, effectively passing every waypoint in a specified radius. Our current policy does not have such feat. Another point for improvement is that the agent is unable to follow trajectories that intersect themselves. This may be resolved by computing the projection only on the 'surrounding' line segments.

References

[1] Ebtehal Turki Alotaibi, Shahad Saleh Alqefari, and Anis Koubaa. Lsar: Multi-uav collaboration for search and rescue missions. *IEEE Access*, 7:55817–55832, 2019.

[2] Bitcraze. Crazyflie Firmware. https://github.com/bitcraze/crazyflie-firmware/, 2024. [Online; accessed 13-Feb-2024].

[3] Adam Bry, Charles Richter, Abraham Bachrach, and Nicholas Roy. Aggressive flight of fixed-wing and quadrotor aircraft in dense indoor environments. *The International Journal of Robotics Research*, 34(7):969–1002, 2015.
[4] Declan Burke, Airlie Chapman, and Iman Shames. Generating minimum-snap quadrotor trajectories really fast. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1487–1492. IEEE, 2020.
[5] Carlos Campos, Richard Elvira, Juan J. Gómez, José M. M. Montiel, and Juan D. Tardós. ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.
[6] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.
[7] Marcelino M De Almeida and Maruthi Akella. New numerically stable solutions for minimum-snap quadcopter aggressive maneuvers. In *2017 American Control Conference (ACC)*, pages 1322–1327. IEEE, 2017.
[8] Marcelino M de Almeida, Rahul Moghe, and Maruthi Akella. Real-time minimum snap trajectory generation for quadcopters: Algorithm speed-up through machine learning. In *2019 International conference on robotics and automation (ICRA)*, pages 683–689. IEEE, 2019.
[9] Dario Floreano and Robert J Wood. Science, technology and the future of small autonomous drones. *nature*, 521(7553):460–466, 2015.
[10] Elia Kaufmann, Leonard Bauersfeld, and Davide Scaramuzza. A benchmark comparison of learned control policies for agile quadrotor flight, 2022.
[11] Connie A Lin, Karishma Shah, Lt Col Cherie Mauntel, and Sachin A Shah. Drone delivery of medications: Review of the landscape and legal considerations. *The Bulletin of the American Society of Hospital Pharmacists*, 75(3):153–158, 2018.
[12] Gregory S McNeal. Drones and the future of aerial surveillance. *Geo. Wash. L. Rev.*, 84:354, 2016.
[13] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE international conference on robotics and automation*, pages 2520–2525. IEEE, 2011.
[14] Jacopo Panerati, Hehui Zheng, SiQi Zhou, James Xu, Amanda Prorok, and Angela P. Schoellig. Learning to fly—a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7512–7519, 2021.
[15] Robert Penicka, Yunlong Song, Elia Kaufmann, and Davide Scaramuzza. Learning minimum-time flight in cluttered environments. *IEEE Robotics and Automation Letters*, 7(3):7209–7216, 2022.
[16] Robert Penicka, Yunlong Song, Elia Kaufmann, and Davide Scaramuzza. Learning minimum-time flight in cluttered environments. *IEEE Robotics and Automation Letters*, 7(3):7209–7216, July 2022.
[17] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.
[18] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023.
[19] EI Verriest and FL Lewis. On the linear quadratic minimum-time problem. *IEEE transactions on automatic control*, 36(7):859–863, 1991.
[20] Zhepei Wang, Hongkai Ye, Chao Xu, and Fei Gao. Generating large-scale trajectories efficiently using double descriptions of polynomials. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7436–7442. IEEE, 2021.

APPENDIX

A. Trajectory Planning

In the domain of autonomous quadcopter navigation, the ability to efficiently generate and optimize trajectories is the foundation for ensuring smooth, safe, and efficient flight. This task comes with inherent complexities because the 12-degrees of freedom (DOF) quadcopter can not move in arbitrary speed along arbitrary sharp curves. Accounting for the quadcopter's dynamics naturally requires an explicit formulation of an optimization objective that integrates the dynamics. However, *differential flatness* eases the integration of the dynamics into trajectory planning. In more particular, the principle implies, that the quadcopter can follow any smooth trajectory in a carefully chosen *flat output space* under the constraints of soundly bounded derivatives [13]. For a detailed explanation of differential flatness refer to appendix A2. More formally, the flat outputs are defined as

$$\sigma = [x, y, z, \psi]^T \tag{4}$$

where $r = [x, y, z]^T$ are coordinates of the center of mass of the quadcopter in the real world coordinate system, and ψ is the yaw angle. The trajectory is now defined in the space of flat outputs:

$$\sigma(t) : [t_0, t_m] \to \mathbb{R}^3 \times SO(2) \tag{5}$$

where t_0 is the start and is the t_m is the end time, and $SO(2)$ refers to the special orthogonal group of degree 2, i.e. the space of all possible rotations around a point in a two-dimensional plane. In this project, we deploy *polynomial minimum jerk/ snap generation* from [20] to generate smooth trajectories $\sigma(t)$ in the flat output space in an fast and efficient manner using [20] and their open-source implementation[3].

1) Minimum Snap/ Jerk Trajectory Generation: This section closely follows the notation of [20] and formalises the used method. Since the spatial dimensions can be decoupled for minimum snap/ jerk polynomial trajectory generation [3] among the four dimensions of σ, we only need to formulate the polynomials and cost functions in one dimension. Define the $(N+1)$-order M-piece spline $(p_1, \ldots, p_M)$ where every spline $p_i, i \in [M]$ is a N-degree polynomial [20]:

$$p_i : [0, T_i] \to \mathbb{R} : t \mapsto c_i^T \beta(t) \tag{6}$$

where $T := (T_1, \ldots, T_M)^T \in \mathbb{R}^M$ is the vector of target timestamps for each piece, $c_i \in \mathbb{R}^{N+1}$ are the coefficients of p_i, and $\beta(t) := (1, t, t^2, \ldots, t^N)^T$ is the natural basis. We get

$$p : [0, \tau_M] \to \mathbb{R} : t \mapsto p_i(t - \tau_{i-1}), \quad t \in [\tau_{i-1}, \tau i] \tag{7}$$

for the entire spline where $\tau_i := \sum_{j=1}^{i} T_j$, and $c = (c_1^T, \ldots, c_M^T)$ as the full coefficient vector of p. Define $s = 3$ for jerk, and $s = 4$ for snap minimization. We set $N = 2s - 1$

[3]https://github.com/ZJU-FAST-Lab/large_scale_traj_optimizer

as optimal degree [19]. The trajectory minimisation problem can now be formulated as follows

$$\min_{c,T} \int_0^{\tau_M} p^{(s)}(t)^2 dt \tag{8}$$
$$s.t. \quad p^{(j)}(0) = d_{0,j}, \quad 0 \leq j < s \qquad \text{(Start derivatives)}$$
$$p^{(j)}(\tau_M) = d_{M,j}, \quad 0 \leq j < s \qquad \text{(End derivatives)}$$
$$p(\tau_i) = q_i, \quad 0 \leq i < M \qquad \text{(Position constraints)}$$

where $d_0 := (d_{0,0}, \ldots, d_{0,s-1})^T$ and $d_M := (d_{M,0}, \ldots, d_{M,s-1})^T$ are initial and final derivatives, respectively, and $q := (q_0, \ldots, q_{M-1})$ are the target waypoints. Note that the minimisation problem takes d_0, d_M, q and T as input, and yields the time-continuous smooth trajectory $\sigma(t)$. For landing, d_M can be set to a zero vector to ensure safe landing. The vector d_0 is dependent on the starting position of the drone, hence a zero vector if the drone starts from a resting position. The target waypoints q need to be obtained through a preceding path planning algorithm such as A^* or RRT*. Also, observe that T defines the timing constraints, and implicitly decides for the aggressiveness of the generated trajectory. To ensure feasibility, velocity and acceleration constraints can be additionally integrated into eq. (8). While there exist a closed-form solution to eq. (8) [3], its computation is inefficient due to the numerical computation of a matrix inverse [20]. We adopt the linear complexity algorithm proposed by Wang et al. [20] which does not necessitate numerically computing the matrix inverse.

2) Differential Flatness: Differential flatness has been validated by [13] and, since then, led to a series of follow-up works [3], [4], [7], [8]. Recall that the trajectory is now defined in the space of flat outputs:

$$\sigma(t) : [t_0, t_m] \rightarrow \mathbb{R}^3 \times SO(2) \tag{9}$$

see eq. (5). To conclude the proof that modeling the trajectory in the flat output space is sufficient for trajectory planning of the underactuated drone, it only needs to be shown that the full state of the system and the control commands sent to the four motors of the drone can be written in terms of $\sigma(t)$. More spefically, the position, velocity, acceleration of the drone, the orientation matrix from the bodyframe to the world frame, the angular velocity and acceleration, and the control motor commands need to be written in terms of $\sigma(t)$. That it is possible to write these variables in terms of $\sigma(t)$ was derived by [13], which builds the foundation of the literature for polynomial trajectory generation. For the full derivation, refer to [13].

B. Full Benchmarks

We compare a traditional position PID controller against our learned policy. Since both control policies take time-discrete waypoints as reference trajectory instead of a time-continuous trajectory, we discretize $\sigma_{land}(t)$ according to different discretization levels [Discr.]. Observe that higher [Discr.] makes both control algorithms result in a slower drone, for different reasons. For the PID controller, higher [Discr.] lets the drone visit and stop at more waypoints (stop-and-go). Tables II and III shows the full benchmarks of this experiment. The PID controller did not achieve 100% success rate for every discretization level due to mis-configured discretization levels. Too low discretization levels observably resulted in to high distance between intermediate waypoints, which made the drone go too fast due to a high proportional componant, resulting in a crash. For too high discretization level, we observed that the drone overshot the target landing location. Note that these issues can be mitigated by tuning the PID parameters. The highlighted rows show a well-configured setting and a fixed time or deviation budget.

Discr.	Model	Success	Mean Dev (m)	Max Dev (m)	Time (s)
5	PID	0.63	0.103	0.372	6.94
6	PID	0.65	0.091	0.320	7.64
7	PID	0.98	0.072	0.228	7.59
8	PID	0.975	0.067	0.187	7.37
9	**PID**	**1.0**	**0.059**	**0.166**	**7.98**
10	PID	1.0	0.055	0.148	8.72
11	PID	0.995	0.053	0.138	9.45
12	PID	0.89	0.051	0.138	10.1
5	RL	1.0	0.141	0.402	5.64
6	RL	1.0	0.119	0.309	5.98
7	RL	1.0	0.104	0.250	6.48
8	RL	1.0	0.090	0.206	7.04
9	RL	1.0	0.082	0.178	7.31
10	**RL**	**1.0**	**0.074**	**0.155**	**7.88**
11	RL	1.0	0.069	0.144	8.41
12	RL	1.0	0.063	0.132	9.08
Avg.	PID	0.89	**0.069**	**0.212**	8.22
Avg.	RL	**1.0**	0.092	0.222	**7.22**

TABLE II: Performance of the trajectory following reinforcement learning (RL) policy on our landing task test set with 200 trajectories. Comparison against a traditional position PID controller.

Discr.	Model	Success	Mean Dev (m)	Max Dev (m)	Time (s)
14	PID	0.37	0.125	0.487	25.66
17	PID	0.545	0.108	0.361	26.3
20	PID	0.9	0.095	0.275	26.72
23	**PID**	**1.0**	**0.089**	**0.228**	**27.02**
26	PID	1.0	0.083	0.209	28.04
14	RL	0.985	0.18	0.49	14.35
17	RL	0.99	0.14	0.39	16.29
20	RL	0.99	0.12	0.31	18.62
23	RL	0.995	0.10	0.27	20.26
26	**RL**	**0.99**	**0.09**	**0.23**	**22.32**
Avg.	PID	0.76	**0.1**	**0.292**	26.75
Avg.	RL	**0.99**	0.126	0.338	**18.37**

TABLE III: Performance of the trajectory following reinforcement learning (RL) policy on our landing task test set with 200 trajectories. Comparison against a traditional position PID controller.