Autonomous Drone Racing – Optimization vs. RL

1st Tim Lindenau
Technical University of Munich
Munich, Germany
tim.lindenau@outlook.de

***Abstract*—Autonomous drone racing has long been researched as a benchmark task for testing control approaches at their limits [1]. While most work focuses on the disturbance-free case, robustness to disturbances in drone dynamics and environmental conditions is important for the real-world deployment of such algorithms. This work presents two approaches that enable autonomous drone racing under disturbances. The first approach is built around subsequent path planning and trajectory generation and demonstrates how, by focusing on computationally efficient implementations, within-flight re-planning is possible and enables high robustness to shifting gates. The second reinforcement learning-based (RL) approach demonstrates that RL can successfully adapt to the domain shifts introduced by disturbances, unifying best-in-class flight times with good robustness levels.**

I. Introduction

This work investigates autonomous drone racing under disturbances. Drone racing is a well-researched topic in control theory as a benchmark task for testing controllers at their limits [1]. The general goal of autonomous drone racing is to design a controller that safely slaloms through a course of gates and additional obstacles in the shortest time possible. Unlike most work which solely focuses on the fastest flight in deterministic environments (e.g., all gate positions known before flight), our investigation is focused on robust drone racing in disturbed environments. Disturbed, here means that drone dynamics, obstacle, and gate positions are only known up to nominal values before the flight, with the true values disturbed by noise. Most importantly, the true obstacle and gate positions are randomly shifted by up to 15 cm, with the true values only becoming available shortly before reaching each gate. Besides the capability of finding fast trajectories, the key factor to succeed in this disturbed environment is the capability of the controller to react to within-flight position updates, re-planning as needed.

We present and compare two approaches capable of achieving fast and robust drone racing in disturbed environments. The first approach is optimization-based and builds on top of the traditional path planning and trajectory generation pipeline, extended with an additional within-flight re-planning component. Robustness to disturbances is achieved by focusing on efficient path planning and trajectory generation algorithms, trading off computational speed for slower flights, and enabling within-flight re-computations around 100 ms. The second approach is based on reinforcement learning, where generalization to disturbed environments is achieved by carefully designing the action and observation spaces, as well as the reward function.

We evaluate and compare both algorithms thoroughly in experiments, demonstrating that both approaches improve baseline performance in terms of mean lap time and success rate. The optimization-based approach excels in robustness, achieving success rates above 80 %. In contrast, the RL agent excels in speed, more than halving the baseline time, despite still achieving good robustness above 70 %.

II. The Challenge Setup

Before presenting both approaches in detail, a short overview of the specific setup around which both are built. At each time step, the controller has access to the current system state $\boldsymbol{s} = [\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{\xi}, \dot{\boldsymbol{\xi}}, \boldsymbol{G}, \boldsymbol{O}, i]$, where $\boldsymbol{x} = [x, y, z]$, $\boldsymbol{\xi} = [\phi, \theta, \psi]$ are the drone position and orientation. $\boldsymbol{G}$, and $\boldsymbol{O}$ are the locations of all gates and obstacles, updated at runtime from nominal to true values, whenever the drone is close enough. i is the index element selecting the next gate.

Based on these observations, the controller must select control inputs. However, instead of directly controlling the drone's motors, the control algorithm is high-level and must provide the desired state of the drone as action $\boldsymbol{a} = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \ddot{x}, \ddot{y}, \ddot{z}, \psi]$. An onboard controller then tracks these commands, converting them into motor thrusts.

III. Optimization Based Drone Racing

The first approach aims for fast drone racing by posing it as an optimization problem of consecutive path planning and trajectory interpolation with an extra re-planning component to adapt to within flight updates. While similar approaches are commonly discussed in the literature [2, 3], most works are not capable of the real-time adaptations required to solve the disturbed environment, recalculating path and trajectory whenever new information becomes available. Our algorithm diverges from previous work by trading off time-optimal solutions for fast computation times, allowing real-time re-computations within 100 ms. This section focuses on the key building blocks, starting with path planning and trajectory generation, before focusing on real-time within-flight re-computations and the involved challenges.

A. Path Planning

Path planning aims to find the time-optimal, collision-free path passing through all gates. Since this path directly depends on the drone dynamics, solving for it is computationally expensive and infeasible within the time-constrained setup that must allow for re-computations. To achieve fast computation

times, we instead employ path length as a proxy criterion, providing much higher computational efficiency through algorithms like *Fast Marching Trees* [4] and *Rapidly Random Exploring Trees* [5]. As our algorithm is indifferent to the choice of path planner, we will not go into deeper discussions on specific planning algorithms but only present the overall idea of utilizing them.

Path planning is initialized with an ordered list of checkpoints $\boldsymbol{C} = [\boldsymbol{c_0}, \boldsymbol{c_1}, ...], \boldsymbol{c_i} \in \mathbb{R}^3$; one for the start, two for each gate position placed before and after the gate center by an offset of δ_{ckpt}, and one for the goal. The racing path is then planned in segments, where pairs of consecutive checkpoints from this list are selected to compute the shortest path between them. For each segment, the path planning process generates a list of waypoints $\boldsymbol{W_i} = [\boldsymbol{w_0}, \boldsymbol{w_1}, ...], \boldsymbol{w_i} \in \mathbb{R}^3$ connected by collision free straight-line segments. Finally, all segments $\boldsymbol{W_i}$ are merged into the overall path $\boldsymbol{W}$.

B. Trajectory Generation

Based on the waypoints $\boldsymbol{W}$, the trajectory generation stage aims to find a time-optimal continuous mapping $\boldsymbol{f}(t) : [0, T] \rightarrow \mathbb{R}^{10}$, from time t to drone control input $\boldsymbol{f}(t)$.

Initially, we experimented using the popular approach of fitting a snap-minimizing trajectory [2]. While successful in previous work, this approach did not work well for us, as the generated trajectory commonly causes collisions by diverging too much from the collision-free straight-line segments of the pre-planned path. This issue is particularly problematic at sharp corners, which frequently occur during re-planning, rendering minimum snap trajectories infeasible for the disturbed environment. Based on this insight, we focused our subsequent approach on guaranteeing path validity while still being computationally efficient and fast, leading us to the now presented two-step pipeline of path smoothing and time parametrization.

1) Path Smoothing: Although the planned shortest-distance path is guaranteed to be collision-free, it suffers from discontinuities at the transition between straight-line segments, hindering the drone from navigating the path at high velocities. Smoothing the path can enhance flight time by enabling higher-speed travel. However, commonly used spline or polynomial smoothing comes at the risk of invalidating the path and is therefore not an option. An alternative approach is found in the work of Sucan et al. [6], proposing an iterative smoothing algorithm that guarantees path validity even after smoothing. A short overview of this algorithm is given below:

During each smoothing iteration, the current path is first subdivided by adding additional points at the midpoint between pairs of waypoints. Then, for each waypoint $\boldsymbol{w}_i$, the previous and next waypoints, $\boldsymbol{w}_{i-1}$ and $\boldsymbol{w}_{i+1}$, are extracted. Two temporary states, $\tilde{\boldsymbol{w}}_{i-1}$ and $\tilde{\boldsymbol{w}}_{i+1}$, are determined as the midpoints between ($\boldsymbol{w}_{i-1}$, $\boldsymbol{w}_i$), and ($\boldsymbol{w}_i$, $\boldsymbol{w}_{i+1}$), respectively. These temporary states are then interpolated to find the new center point $\tilde{\boldsymbol{w}}_i$. If the straight-line connection from $\boldsymbol{w}_{i-1}$ to $\tilde{\boldsymbol{w}}_i$ and from $\boldsymbol{w}_{i+1}$ to $\tilde{\boldsymbol{w}}_i$ are valid, $\boldsymbol{w}_i$ is replaced with $\tilde{\boldsymbol{w}}_i$ before continuing the loop.

Repeating this process over multiple iterations significantly increases both the number of waypoints and the smoothness of the path, while still guaranteeing overall path validity.

2) Path Parametrization: The second path parametrization (PP) step concludes trajectory generation, by solving for the time optimal control mapping $\boldsymbol{f}$ of the smoothed path.

While significantly increased in the number of waypoints, the smoothed path is not yet continuous and must first be interpolated into $\boldsymbol{\gamma}(\tau) : [0, 1] \rightarrow \mathbb{R}^3$ with τ the path progress parameter. It should be noted that general interpolation approaches, such as splines or circular blends, can now be used without risking path invalidity, as the additional waypoints introduced during smoothing provide enough constraints to prevent overshooting. The goal of PP is then to find a fast reparametrization $\nu : [0, T] \rightarrow [0, 1]$ to pass $\boldsymbol{\gamma}$ as fast as possible. Velocity, acceleration, and heading can finally be derived by taking the time derivative of $\boldsymbol{\gamma}(\nu(t))$.

Although simple and effective for the winners of the last *IROS2022 Safe Robot Learning Competition* [7], parametrizing the path based on uniform path velocity $\dot{\tau}$ or similar velocity profiles like S-Curves cannot provide the best performance on a general racing track with sharp corners. This should be intuitive, since on straight-line segments the optimal parametrization should pass the path much faster (high $\dot{\tau}$) than in sharp corners (low $\dot{\tau}$). A better and provable time-optimal parametrization technique is provided in the work of Kunz and Stilman [8], proposing the *Time-Optimal Trajectory Generation for Path Following with Bounded Acceleration and Velocity* (TOTG) method. TOTG is an efficient algorithm to find the provable time-optimal parametrization taking into consideration the drone dynamic limits, as well as the shape of $\boldsymbol{\gamma}$. A high-level introduction to TOTG is given in the following. The interested reader is referred to [8] for a more detailed treatment.

At each position τ the velocity and acceleration of the drone subject to the path $\boldsymbol{\gamma}$ can be calculated as

$$\dot{\boldsymbol{x}} = \frac{\mathrm{d}}{\mathrm{dt}}\boldsymbol{\gamma}(\tau) = \frac{\mathrm{d}}{\mathrm{d}\tau}\boldsymbol{\gamma}(\tau)\frac{\mathrm{d}\tau}{\mathrm{dt}} = \boldsymbol{\gamma}'(\tau)\dot{\tau} \tag{1}$$

$$\ddot{\boldsymbol{x}} = \boldsymbol{\gamma}'(\tau)\ddot{\tau} + \boldsymbol{\gamma}''(\tau)\dot{\tau}^2 \tag{2}$$

Instead of requiring the full dynamics of the drone, TOTG assumes bounded velocity $\dot{\boldsymbol{x}}_{\text{max}}$ and acceleration limits, $\ddot{\boldsymbol{x}}_{\text{max}}$ allowing to formulate path velocity $\dot{\tau}$ and path acceleration $\ddot{\tau}$ constraints at each position as a function of τ and $(\tau, \dot{\tau})$.

Naturally, to move along $\boldsymbol{\gamma}$ as fast as possible, the optimal controller is a bang-bang controller only switching between the maximum possible path acceleration $\ddot{\tau}_{\text{max}}(\tau, \dot{\tau})$ and minimum acceleration $\ddot{\tau}_{\text{min}}(\tau, \dot{\tau})$. The goal of TOTG is to find the optimal switching points between these control inputs, which is efficiently solved in a process of forward and backward integration. During forward integration, $\boldsymbol{\gamma}$ is progressed with maximum acceleration until reaching a limit constraint. During backward integration, the same process is repeated but starting from the back of $\boldsymbol{\gamma}$ and using minimum acceleration. Switching points are then identified as the intersection of forward-

and backward passes. A deeper understanding of this algorithm and its challenges is provided in this Youtube tutorial, as well as section VI of Kunz and Stilman [8].

C. Challenges of Going Online

The previous sections described the general process of path planning and trajectory generation. This section focuses on the additional challenges introduced by re-planning to account for updated gate positions delivered within flight. Re-planning for obstacle updates is not yet implemented, however, some potential benefits are discussed in Section VI-D.

The key challenge in re-planning is fast computation time and non-blocking execution. We address the need for fast execution through our emphasis on runtime-efficient path planning and trajectory generation, all implemented in fast, multithreaded C++. Furthermore, as an update in the next gate position can only influence the path from the current drone position to this gate, as well as from the updated gate to the next gate, a further improvement by $2/N$, N being the total number of gates, can be trivially achieved by only re-planning these two segment

While these measures enable the reduction of the re-planning runtime from several seconds in a pure Python implementation to only approximately 100 ms in C++, this duration remains too long for straightforward integration of re-planning and motivates the necessity for non-blocking execution. To illustrate, consider a drone moving towards a goal at a realistic speed of 2 m/s. Even with a re-planning time of only 100 ms, the drone will move 20 cm during this period without any additional control input to constrain its movement. A non-blocking execution, combined with an offset to the start-position of re-planing, can effectively overcome this challenge. Non-blocking execution means that re-planning is executed in a thread separate from the main control loop, allowing the controller to re-use the previous trajectory while re-planning and keeping the drone's position predictable on this trajectory. With the motion of the drone predictable on the current trajectory, offsetting the initial starting point of re-planning based on the expected computation time along the previous trajectory enables smooth merging between the previous and re-planned trajectory.

IV. Reinforcement Learning Based Drone Racing

Optimization-based techniques like ours have long been the leading approach for autonomous drone racing [1]. Yet, recent advancements in reinforcement learning (RL) have shown remarkable performance, often surpassing optimization methods and even human capabilities in some cases [9, 10]. Compared to optimization methods, the primary advantage of reinforcement learning (RL) is its ability to learn an end-to-end control policy directly from training examples, instead of requiring hand-crafted and potentially suboptimal components.

For instance, in optimization-based drone racing, various factors must be balanced, choosing between different path and trajectory planners based on considerations such as time optimality and computational complexity. Our path planning algorithm, for example, is not time-optimal due to its reliance on a minimum distance proxy. In contrast, by encapsulating logic in the neural network, which can be efficiently evaluated in just one forward pass per time step, the RL-drone can learn optimal control policies without incurring the same tradeoffs. However, RL also has known disadvantages, most notably its limited generalizability to domain shifts. In disturbed environments, each new realization can be considered a domain shift that might be challenging to overcome.

Through careful design of the reward, action- and observation space, we are able to train an RL agent that successfully generalizes to the dynamic environment, outperforming the optimization approach in terms of speed while maintaining good robustness.

A. Task Formulation

Song et al. [10] have illustrated how drone racing can be formulated within the traditional RL framework as an infinite-horizon Markov Decision Process MDP [11], described by the tupel $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$. Based on the current state $\boldsymbol{s}_t \in \mathcal{S}$ an action $\boldsymbol{a}_t \in \mathcal{A}$ is sampled from the stochastic policy $\pi\left(\boldsymbol{a}_t|\boldsymbol{s}_t\right)$, transitioning the agent to the next state $\boldsymbol{s}_{t+1}$ dependend on the state-transition probability $\mathcal{P}^{\boldsymbol{a}_t}_{\boldsymbol{s}_t,\boldsymbol{s}_{t+1}} = \Pr\left(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t\right)$. Each state transition is accompanied by an associated reward $r(t) \in \mathbb{R}$ to encourage desirable behavior.

Deep RL aims "to optimize the parameters $\boldsymbol{\theta}$ of a neural network policy π_θ such that the trained policy maximizes the expected discounted reward over an infinite horizon

$$\pi^*_{\boldsymbol{\theta}} = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t r(t)\right] \tag{3}$$

where $\gamma \in [0, 1)$ is the discount factor" [10] and $\boldsymbol{\tau}$ the realization of a trajectory according to the policy π.

In the following sections, we motivate our choices for the reward function r, state space $\mathcal{S}$, and action space $\mathcal{A}$, which enable successful drone racing even in the presence of disturbances.

1) Action- and Observation Space: Designing an effective action and observation space proved to be the most important factor for successful flight in disturbed environments in our experiments. Two factors, in particular, demonstrated significant importance and are discussed in the following sections.

Simplifying the action and observation spaces to avoid unnecessary information is crucial for achieving good performance. We simplify both spaces by first removing all yaw information from them. This is motivated as yaw is not necessary to properly control or stabilize the drone and instead can just be fixed to 0. Complexity is further reduced, by filtering out gate and obstacle positions, only providing position information about the next gate and closest obstacle to the RL agent. This is motivated by initial experiments, showing that when providing the positions of all gates, the drone is not able to successfully identify the next gate and therefore unable to react to within-flight updates. Instead, the drone just overfits the nominal track which still provides good performance in the static environment, however, completely

fails with disturbances. While not providing all gate information could potentially hinder the drone from learning the globally optimal policy, this did not have a significant negative impact on performance in our experiments and was accepted as a well-worth tradeoff.

Translating relevant information into the local coordinate system of the drone is the second performance-unlocking factor. In the observation space, positional information about the obstacles and gates is provided in coordinates relative to the drone. The action space is relative as well, providing commands as offsets from the current position instead of locations in the global space. Providing relative information is useful because it reduces the complexity of the policy the drone has to learn. Instead of having to learn a policy taking into consideration the whole global constellation between the drone and all gates, this way, the drone only has to learn once how to pass a gate depending on its relative position, which then generalizes.

Applying both steps, the observation space becomes

$$\boldsymbol{s}_{\text{RL}} = [x, y, z, \phi, \theta, \tilde{\dot{x}}, \tilde{\dot{y}}, \tilde{\dot{z}}, \dot{\phi}, \dot{\theta}, \tilde{\boldsymbol{g}}, \tilde{\boldsymbol{o}}], \tag{4}$$

where $\boldsymbol{g}$ is a vector containing the 4 corner positions of the next gate and $\boldsymbol{o}$ is a vector containing the position of the nearest obstacle. The $\tilde{\cdot}$ operation denotes variables whose coordinates are given in local frames corresponding to the current drone orientation $[x, y, z, \psi]$.

The action space is given as $\boldsymbol{a}_{\text{RL}} = [\Delta x, \Delta y, \Delta z] \in [-S_{\text{action}}, S_{\text{action}}] \times [-S_{\text{action}}, S_{\text{action}}] \times [-S_{\text{action}}, S_{\text{action}}]$, with S_{action} called the action bound. The conversion into global coordinates is achieved as

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix}, \tag{5}$$

which are then converted into $\boldsymbol{a} = [a_x, a_y, a_z, \mathbf{0}^T]$, to align with the format expected by the onboard controller.

2) Reward Function: Similarly to [9, 10], we employ per time-step progress to the next gate as the main reward. Given the current and previous drone position $\boldsymbol{x}(t)$, $\boldsymbol{x}(t-1) \in \mathbb{R}^3$, as well as the center position of the current gate $\boldsymbol{g} \in \mathbb{R}^3$, the progress reward is defined as

$$r_{\text{progress}}(t) = \|\boldsymbol{x}(t-1) - \boldsymbol{g}\|_2 - \|\boldsymbol{x}(t) - \boldsymbol{g}\|_2 . \tag{6}$$

This reward acts as a good proxy for the objective of race time while providing much denser supervision at each time step.

Binary rewards for passing a gate $r_{\text{gate}}(t)$, as well as for crashing $r_{\text{col}}(t)$ complete the overall reward function

$$\begin{aligned} r(t) = \lambda_{\text{progress}} r_{\text{progress}}(t) + \lambda_{\text{velocity}} r_{\text{velocity}}(t) \\ + \lambda_{\text{gate passed}} r_{\text{gate}}(t) + \lambda_{\text{col}} r_{\text{col}}. \end{aligned} \tag{7}$$

V. Results

A. Experimental Setup

Experiments are conducted for both approaches using the specific parameter choices described in Appendix A. A manually tuned trajectory, created by placing waypoints along the racetrack with a uniform path velocity, is used as the baseline for comparison. All approaches are evaluated for speed and robustness in the static and disturbed environment, the static environment showing the upper bound on speed each approach can achieve, and the disturbed environment showing robustness and the capability for re-planning. To guarantee reproducibility, all experiments are repeated over 100 flights, with the results averaged among them.

B. Comparison Optimization and Reinforcement Learning

Table I and Figure 1 summarize the performance achieved by the different approaches with and without disturbances. In the static environment, both algorithms demonstrate their capability of finding fast racing trajectories, with the optimization-based approach almost twice as fast as the baseline and the RL approach even faster reducing the baseline-lap time by $60\,\%$. Despite being fast, both approaches are still robust achieving high success rates of $86\,\%$ and $100\,\%$.

While the hard-coded baseline fails in the disturbed environment with a success rate of only $18\,\%$, both of our approaches significantly improve robustness. The optimization-based approach, in particular, shows a high degree of robustness, achieving a success rate of $81\,\%$. Unfortunately, this high success rate can only be achieved at the cost of significantly slowing down the drone due to two factors. First, the re-planning operation often introduces sharp turns that the drone cannot follow at higher speeds. Second, when shifted by disturbances, the constellations between gates and obstacles potentially become much tighter than on the nominal track. This gives less tolerance for overshoots which are common at high speeds, further necessitating a speed reduction. More information about overshoots is provided in the next section.

The RL agent cannot quite achieve the same degree of robustness; however, it is much faster, incurring only a $0.2\,\text{s}$ drop in runtime to $5.17\,\text{s}$. Although it does not reach the optimization level, the $71\,\%$ success rate demonstrates that the RL agent successfully learns to react to environment changes and does not just remember one racing line by heart – an issue long prevalent in initial experiments.

Figure 5 helps to better understand how, in the static environment, the RL agent can outperform the optimization-based approach in three key ways. The first two figures show the true racing trajectories of both approaches collected over 100 trials. The third figure is a side-by-side comparison of optimization and RL using one exemplary trajectory.

The first performance explaining factor is that the optimization drone commonly overshoots the planned trajectory at sharp turns. (1) in Figure 5 illustrates this behavior well. While both approaches attempt the same maneuver – a sharp stop within the gate – the optimization-based drone significantly overshoots this stop. In theory, overshoots should be rare since the trajectory generator already provides the drone's position, velocity, and acceleration at each time step to guarantee the feasibility of future states. One potential reason why overshoots still appear is the indirect control with an onboard controller incapable of tracking the given trajectory.

Algorithm	Difficulty	Mean Time (s)	Std. Time (s)	Success Rate (%)
Baseline	static	12.47	0.00	100
Baseline	disturbed	12.47	0.09	18
Optimization	static	6.50	0.36	86
RL	static	**4.97**	**0.01**	**100**
Optimization	disturbed	9.78	0.74	**81**
RL	disturbed	**5.17**	0.33	71

TABLE I: Performance comparison reinforcement learning vs. optimization. All results averaged over 100 runs.

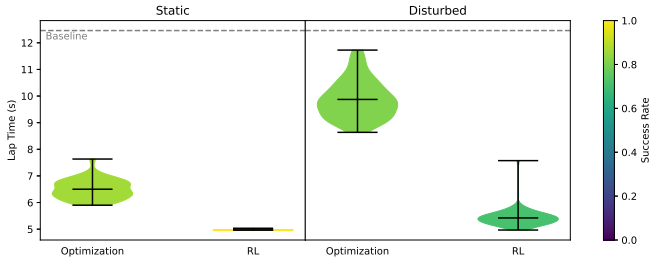

Fig. 1: Variance analysis reinforcement learning vs. optimization. All results averaged over 100 runs.

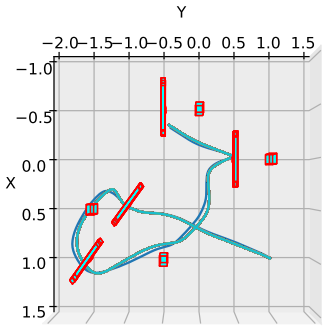

Fig. 2: Reinforcement Learning

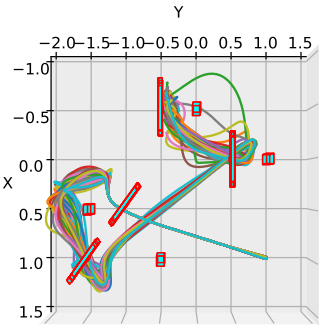

Fig. 3: Optimization

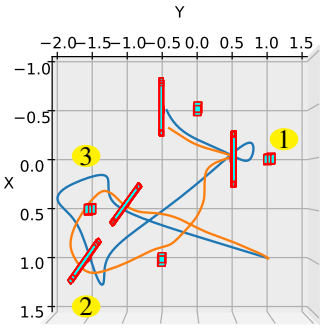

Fig. 4: Optim. (blue) vs. RL (orange)

Fig. 5: Collection of flight trajectories over 100 runs for the static environment.

In contrast, the RL agent implicitly learns the system dynamics including the onboard controller, thereby reducing overshoots.

The second factor, closely related to the first, is the overall optimization criterion of minimizing path length. As discussed before, this criterion is only a proxy for time-optimal flight. In practice, it suffers from causing trajectories with sharp turns, increasing the risk of overshoots and forcing the drone to slow down significantly. As ② of Figure 5 illustrates, a different optimization criterion could enhance performance by trading path length for reduced curvature, thus increasing path velocity, minimizing overshoots, and overall reducing flight time.

The third factor contributing to the faster times of the RL agent is the various adaptations necessary within the optimization-based approach to guarantee safe flights, even in the presence of issues such as overshoots. One such adaptation is the requirement to manually inflate all gates and obstacles by 15 cm, a precaution the RL agent does not need. ③ in Figure 5 is a good example where the RL agent can shortcut significantly by passing the obstacle very closely.

Finally, analyzing the optimization trajectories in Figure 3 also explains the comparatively high variance of the optimization-based approach. While in the disturbed environment all approaches naturally show variance in their flight times, the optimization-based approach already shows significant variance in the static environment. We believe that this variance stems from the time constraints we impose on the path planner, allowing it to only find an approximately shortest path that diverges between each run. Experimenting with different maximum planning times for path planning validates this intuition, as variance increases inversely with planning time. For best performance, it is therefore recommended to set the maximum computation time as long as possible, depending on hardware and track layout, and to use longer computation times for pre-planning to optimize the initial path.

VI. Ablations

A. Improving Reinforcement Learning Robustness

While the RL agent already provides good robustness, flight analysis reveals a common error responsible for most crashes. Since the observation space only provides information about the next gate, the currently passed gate becomes invisible to the drone after passing. As a result, the drone frequently crashes into the gate it just passed when navigating toward the next gate.

To overcome this issue, we experimented with including the position of the closest gate as an additional observation. Unfortunately, this did not have the desired effect. The same error case still prevailed and success rates dropped below 60 %.

B. Potential Speed Improvements by Preventing Direction Changes Within Gates

Analyzing the trajectories in Figure 5, one can observe that most of them opt for a sharp turn within the third gate, instead of flying around it. For the minimum distance path planner, this result is expected as it represents the shortest path. However, it raises the question of whether coming to a full stop is faster

Algorithm	Difficulty	Mean Time	Std. Time	Success Rate
MST	static	15.91	0.46	0.8
MST	static	10.67	**0.21**	0.55
Ours	static	**6.50**	0.36	**0.86**
MST	disturbed	17.18	1.12	0.4
Ours	disturbed	**9.78**	**0.74**	**0.81**

TABLE II: Performance comparison minimum snap trajectory generation.

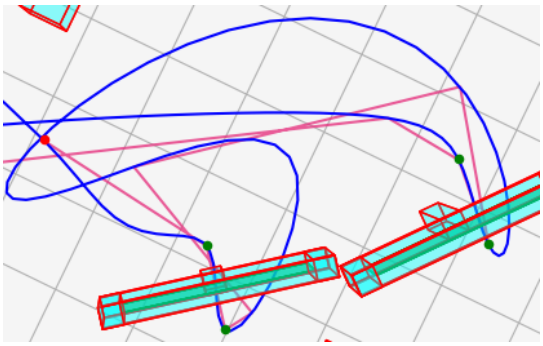

Fig. 6: In an attempt to minimize snap, the trajectory (blue) gets overly complicated, causing slow flight and collisions. The pre-planned straight-line path is shown in red.

than taking the slightly longer path around the gate, which might allow for higher velocities. To investigate, we conduct an additional experiment where the drone is prohibited from passing a gate again after the initial pass. Compared to the original static environment implementation, this leads to an average lap time increase of 1.28 seconds.

C. Performance Comparison Minimum Snap Trajectory

Many approaches for generating a trajectory from waypoints exist. We show the effectiveness of our approach compared to the widely used minimum snap trajectory generation [2] (MST). MST is a popular choice for drone racing as it has empirically been shown to generate trajectories imitating human drone pilots. In our setup, however, MST did not provide good performance. Table II collects the flight times of MST[1] compared to our optimization approach. While MST manages safe-drone flight at low speeds in the static environment, it is incapable of matching the performance of our approach at higher speeds. Furthermore, it is especially struggling with gate shifts and re-planning, only achieving a success rate of 40 %, even at very low speeds.

An empirical analysis reveals two common problems with the MST algorithm. First, unlike our approach, MST does not incorporate any world representation. Consequently, it often generates trajectories that diverge too much from the collision-free straight-line segments of the pre-planned path, thereby causing collisions. Second, in its attempt to minimize snap at no other cost, MST trajectories often end up significantly more complex than the original path. This is well illustrated in Figure 6, where MST overshoots the path, introducing many more turns than necessary, which the drone is unable to follow at higher speeds. As sharp turns within the path exaggerate the latter issue, it becomes especially problematic when we do re-planning, explaining the performance drop in the disturbed environment.

[1]Implemented according to https://github.com/ethz-asl/mav_trajectory_generation.

D. Potential Benefit of Incorporating Obstacle Updates

The optimization algorithm currently incorporates updates only for gate positions, not obstacle positions. To measure the negative impact, we test our approach in an environment where the gates still move but all obstacles are fixed, leading to an increase of the success probability by 9 % points to 90 %. Incorporating obstacles further allows for a decrease in flight time, by facilitating scaling down the inflation radius. For example, scaling down the inflation radius to 10 cm leads to a 0.70 s decrease in mean-time to 9.07 s, while still being more robust than the original optimization approach, achieving a success rate of 86 %. This result emphasizes the importance of including gate updates both for robustness and reduced flight time in future iterations of our work.

VII. From Simulation To Real World

We successfully transferred the optimization-based approach from simulation to the real world. To achieve this, a slight decrease in velocity and acceleration was necessary due to the dynamic mismatch between the simulation and the real world. With this adaptation, we achieved flight times between 8 s and 9 s in the static environment, around 2 s slower than in simulation. Reacting to gate shifts has not been tested in the real world due to limitations in the deployment software stack. Despite this, we remain optimistic about the feasibility of such a transfer, given our focus on non-blocking, fast execution.

Deploying any RL agent to the real world is generally challenging. We expect the same to be true, however, we do not have any results due to missing time.

VIII. Conclusion

In this work we presented and compared two approaches for solving autonomous drone racing in disturbed environments. By focusing on efficient implementation, the optimization-based algorithm is capable of within-flight re-planning and achieved the best robustness to disturbances. Additionally, being built around general-purpose components without requiring per-racetrack tuning, it is likely capable of achieving this performance in diverse environments, a potentially important factor for real-world utilization. However, while still significantly faster than the baseline, it struggled to achieve very fast flight, being more than 4.6 seconds slower than the RL agent in the disturbed environment.

The RL agent, on the other hand, demonstrated consistently impressive performance, achieving the fastest flight times by a significant margin, while maintaining high robustness. We are optimistic that by addressing the identified error cases of the RL agent, its robustness can be further improved, potentially surpassing that of the optimization-based approach.

References

[1] D. Hanover, A. Loquercio, L. Bauersfeld, A. Romero, R. Penicka, Y. Song, G. Cioffi, E. Kaufmann, and D. Scaramuzza, "Autonomous Drone Racing: A Survey," Jan. 2023.

[2] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 2520–2525.

[3] Z. Wang, X. Zhou, C. Xu, and F. Gao, "Geometrically Constrained Trajectory Optimization for Multicopters," Apr. 2022.

[4] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," vol. 34, no. 7, pp. 883–921. [Online]. Available: https://doi.org/10.1177/0278364915577958

[5] S. Karaman and E. Frazzoli. Sampling-based Algorithms for Optimal Motion Planning. [Online]. Available: http://arxiv.org/abs/1105.1186

[6] I. A. Sucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," vol. 19, no. 4, pp. 72–82. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6377468

[7] "IROS 2022 Safe Robot Learning Competition Awards." [Online]. Available: https://www.youtube.com/watch?v=-il6B1XeJkI

[8] T. Kunz and M. Stilman, "Time-Optimal Trajectory Generation for Path Following with Bounded Acceleration and Velocity," in *Robotics*, N. Roy, P. Newman, and S. Srinivasa, Eds. The MIT Press, pp. 209–216. [Online]. Available: https://direct.mit.edu/books/book/3987/chapter/166309/Time-Optimal-Trajectory-Generation-for-Path

[9] Champion-level drone racing using deep reinforcement learning — Nature. [Online]. Available: https://www.nature.com/articles/s41586-023-06419-4

[10] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous Drone Racing with Deep Reinforcement Learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 1205–1212. [Online]. Available: https://ieeexplore.ieee.org/document/9636053/

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press.

[12] Z. Yuan, A. W. Hall, S. Zhou, L. Brunke, M. Greeff, J. Panerati, and A. P. Schoellig. Safe-control-gym: A Unified Benchmark Suite for Safe Learning-based Control and Reinforcement Learning in Robotics. [Online]. Available: http://arxiv.org/abs/2109.06325

[13] W. Giernacki, M. SkwierczyÅski, W. Witwicki, P. WroÅski, and P. Kozierski, "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering," in *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, pp. 37–42. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8046794

[14] DLR-RM/stable-baselines3: PyTorch version of Stable Baselines, reliable implementations of reinforcement learning algorithms. [Online]. Available: https://github.com/DLR-RM/stable-baselines3

Appendix

A. Experimental Setup

All experiments are conducted using the *safe-control-gym* [12] benchmarking environment with the *Crazyflie 2.0* quadrotor [13] as the dynamic model.

For all optimization-based experiments the checkpoint offset to gates, as well as the inflation radius around gates and obstacles is fixed to $\delta_{\text{ckpt}} = 0.15\,\text{m}$. For path planning, we are using the *Open Motion Planning* [6] C++ library, due to its efficient implementation of many common path planners. The *Fast Marching Tree* (FMT) [4] path planning algorithm is utilized due to it being consistently able to outperform all other approaches in experiments. The number of samples for FMT is fixed to 10,000, this being a good trade-off between computation speed and performance, enabling the re-planning to happen within 100 ms. Maximum acceleration and velocity are custom-tuned per experiment through a hyperparameter study, always aiming for a success rate above $80\,\%$. In the static environment, this delivered $v_{\text{max}} = 4.02\,\text{m/s}$ and $a_{\text{max}} = 3.19\,\text{m/s}^2$. For the dynamic environment $v_{\text{max}} = 3.6\,\text{m/s}$ and $a_{\text{max}} = 1.5\,\text{m/s}^2$.

The reinforcement learning agents are implemented using *stable baselines 3* [14]. *Proximal Policy Iteration* (PPO) is utilized as the learning algorithm due to its robustness to changes in hyperparameters and generally good performance. Furthermore, we are using $\lambda_{\text{progress}} = 3$, $\lambda_{\text{col}} = 1$, $\lambda_{\text{gate}} = 3$, and $S_{\text{action}} = 1$. As diverse experiences help RL to learn faster, we randomly initialize the drone in different gates during training.

B. Code

All code is provided in the following GitHub repositories, with installation instructions provided in the READMEs.

1) Optimization-Based Drone Racing:

- Efficient Path Planner: C++ code for path planning and trajectory generation.
- LSY Drone Racing: Python controller to interact with the simulation environment. The C++ code is connected with a Python binding.

2) Reinforcement Learning:

- LSY Drone Racing: Training code for the RL agent, as well as code for the controller to integrate with the simulation environment.
- Safe Control Gym RL: Minor adaptations to the original code to allow randomization of drone-starting positions, as well as enabling curriculum learning, continuously increasing the environment disturbances depending on the drone's performance.

C. Video

Videos of the drone flying in simulation and the real world are provided online.