Practical Course Report: Landing Drones

Daniel Bin Schmid, Matej Straka, Alberto Confente, Daniel He,
School of Computation, Information, Technology
Technical University of Munich
{danielbin.schmid,matej.straka,daniel.he}@tum.de
ge84cuf@mytum.de

***Abstract*—In this project, we gain hands-on experience for working with unmanned area vehicles (UAVs) by building and implementing the infrastructure and software for a Crazyflie. We operate in the realm of a landing scenario without an external tracking system for getting an estimate of the quadcopter state. More specifically, the state of the drone is solely estimated from the IMU and RGB data collected on-board of the UAV using ORBSLAM3. In addition to addressing real-world deployment, we experiment with reinforcement learning (RL) based and model-based control algorithms. In particular, we experiment with a trajectory following RL policy from drone racing and implement model-predictive control using CASADI. 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, State Estimation, Trajectory Planning, Reinforcement Learning, Model Predictive Control, Vision-based Navigation, Simulation and Real-world Deployment**

I. Introduction

Small drones, or also unmanned aerial vehicles (UAVs), are predicted to have wide socio-economic impact once the legal roadblock is lifted [14]. They can be used in a wide array of applications such as for security and surveillance [20], aiding in rescue operations [2], or delivery and logistic [18]. In this project, we work hands-on with one of such small UAVs, namely Crazyflie (see fig. 1), and has two main focuses. First, we implement a full pipeline from vision down to control using ROS2 [19] without depending on an external position tracking system. For our second focus, we experiment with reinforcement learning (RL) based control approaches from drone racing [25], which we implement using an open-source simulator [24] that connects pybullet physics simulation [9] with Gymnasium [29] and stable-baselines [27], and Bitcraze's Crazyflie firmware [4].

The objective of this project evolves around the application of *landing*, which led to design decisions such as placing the onboard camera on the bottom of the UAV or in designing test suites for our RL based control agent. The scope of this project is not to implement a state-of-the-art landing software, but to gain hands-on experience with all the components involved, connect hardware pieces including their interconnectivity onto the plain Crazyflie which did not have a camera on-board in its initial state, understand the signficance of the encountered problems when moving from the simulation to the real-world, and to follow individual interests in specific algorithms.

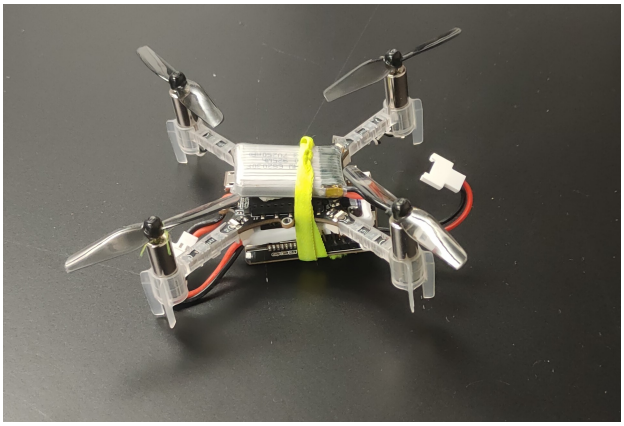


Fig. 1: Our Crazyflie.

This report is structured as follows. Section II provides an overview over the implementation details. The subsequent sections dive into individual focuses, including state estimation in section III, trajectory planning in section IV, model predictive controller (MPC) in section V, and RL in section VI. In section VIII, a conclusion of the project is given.

II. Implementation

We operate in three environments, i.e. two simulation environments and real-life. We use Webots [22] and its integration with ROS2 and `gym-pybullets-drones` [24] for the RL experiments because both environments already provide built-in integration of the Crazyflie. The software written for Webots is the same software that we also use for the real-life scenario.

A. ROS2 Software

Figure 2 gives an overview of our ROS2 software. It is structured in three components, detection and state estimation, trajectory planning, and control. The pipeline takes in data collected on the drone as input, i.e. data from the internal measurement unit (IMU) and RGB camera data. It outputs control commands, which are in turn sent to the drone. On the Crazyflie, we mount an ESP32-camera [13], from where

the camera data is streamed. During simulation, the interfaces to the real drone are replaced by interfaces to Webots.

1) ROS2 Design: Our software entails 6 packages, `teleop`, `ros2_driver`, `localization`, `trajectory`, `detectors`, and `controllers` (each with a '`crazyflie_`' prefix). An additional package defines custom messages and another package defines global launch files. Each package features spawning ROS2 nodes and implements communication via standard ROS2 primitives, i.e. topics, services and actions. The code is provided on github[1].

a) Teleoperation: The `teleop` package implements the interface to external controllers, and publishes converted signals to other ROS2 topics. We implement the interface to a Dualshock PS3/PS4 controller [28], and convert the I/O signals to attitude commands. The interface allowed for straightforward debugging of other components.

b) Control Interface: The `ros2_driver` package implements the interface to the Crazyflie and Webots, i.e. sending motor-level pulse width modulation (PWM) signals or attitude commands. The interface to the Crazyflie is implemented using cflib[2].

c) Localization: The `localization` package is responsible for state estimation, which is further detailed in section III.

d) Detection: We detect landing targets using AprilTag [23]. The interface is implemented in `detection`, which publishes the outputs of the AprilTag software to a ROS2 topic. The AprilTags are also integrated into Webots, enabling debugging with AprilTags in the simulation.

e) Trajectory Planning: Converting the current state and target location to a feasible trajectory is implemented in the `trajectory` package. More details can be found in section IV. Note that within the scope of the project, this component was not used because other components were deemed to be more significant to implement a minimum viable product (MVP). Especially a stable state estimation and video stream are crucial to implement simple hovering or waypoint tasks, which do not necessitate sophisticated trajectory planning. For instance, for the latter task, trajectory planning reduces to specifying a single target waypoint, which can be done without a dedicated package.

f) Control: The `controllers` package implements PID control in C++. We opted for a two-stage approach, where one stage converts attitude to low-level PWM commands, and the other stage converts position to attitude commands. More about this is in appendix D.

B. Hardware Tasks

The project involved working closely with the hardware and this section provides details about the tasks that came with it.

1) Hardware Build Versions: In the initial state, we received a disassembled Crazyflie 2.1 and a plain standalone ESP32 camera. Assembling the Crazyflie itself is straightforward but there was no out-of-the-box solution for mounting the camera. The first version with the ESP32 camera on-board involved soldering pins of the camera together with pins of the Crazyflie, which was done by the supervisor of the project. This solution, however, posed the issue that the propeller motors and the camera were connected to the same battery. Due to insufficient power supply, this resulted in a shutdown of the camera stream once the motors were turned on. We believe that the cause for this behaviour is that the radio module on-board is shutdown once the power-supply falls below a threshold due to a security mechanism. As a solution, we added a separate battery onto the drone. However, the second battery increased the weight of the UAV to a degree where the motors were too weak to lift the Crazyflie. To counteract too weak motors, we upgraded the drone to stronger motors, which, however, significantly reduced the flight time to about one to two minutes. The final build version is shown in fig. 1 with the camera facing downwards.

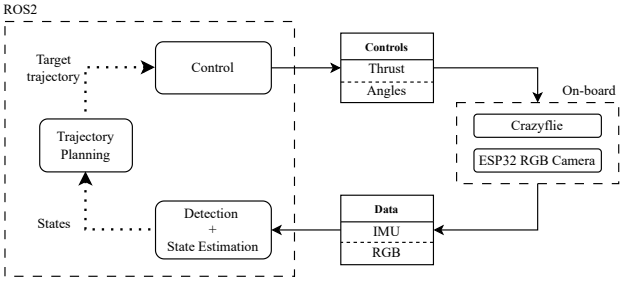


Fig. 2: Overview of our ROS2 software.

2) Improving the Video Stream: Since the state estimation implementation we adopted is solely based on the RGB and IMU data from the drone (see section III), a stable video stream presents a foundational component of our software. However, we found that the video stream via WiFi was very unstable and disrupted frequently. A description of the approached solution using packet injection can be found in appendix C. A deep dive into open-source code for this approach revealed that this task is out of the scope of this project, and left up to future work.

C. Pybullet Simulation

The simulation environment `gym-pybullet-drones` [24] significantly eased experimentation because it sped up simulation speed from $2\times$ in Webots 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 [16]. Furthermore, minimum-snap polynomial trajectory generation from section IV written in C++ is integrated by implementing an interface to python via `pybind11`. The code is made available on github[3].

[1]https://github.com/AlboAlby00/CrazyflieControllers/

[2]https://github.com/bitcraze/crazyflie-lib-python

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

D. Open Problems

The encountered issues prevented us from finalising our ROS2 software. The key restriction is the *unstable state estimation* combined with the *unstable video stream*, plus the issues of our improvised hardware build solution explained in section II-B1. Implementing a simple position control requires a stable estimate of the own position in the world coordinate frame. Without a stable estimate, translating high level position commands from a reference trajectory in the world coordinate system to attitude commands in the bodyframe coordinate system, is ill-posed. While AprilTag gives directions to a target location in the bodyframe coordinate system, it requires a stable video stream, which was not given. In addition, preparing the UAV for every planned flight introduced significant overhead due to battery charging and camera mounting preparation times. On average, we prepared our drone for 20 minutes for one minute flight time.

III. State Estimation

The whole state estimation part of the project was developed in simulation using Webots, on Pybullet we used directly the ground truth provided by the simulator. The state estimation part using only RGB data is also working in the real world, but the stability is impacted by the bad video stream. For the task of estimating the state of the robot, the only data available was the angular velocity, linear acceleration and orientation obtained from the IMU inside the drone, and RGB data from an ESP32 camera. After a careful review of the possible algorithms to use, we opted for ORBSLAM3. ORBSLAM3 is a indirect visual SLAM algorithm that can be used with monocular, stereo or RGBD cameras. In the monocular setup it extracts features from two suitable frames, computes the correspondences and project them in 3D by triangulation. 2D-3D correspondences from following frames and the 3D map are then used to estimate the translation and rotation from the current frame to the map frame, and 2D features not already in the 3D map are projected and added to it. Monocular Visual SLAM is not able to estimate the true scale of the translation, so the final trajectory is correct up to a constant scalar value that is not possible to obtain only using RGB data. Using depth (that we don't have) or inertial data (that we have), it is theoretically possible to estimate correct translations. ORBSLAM3 provides also a visual inertial monocular mode, but the stability was substantially worse compared to the monocular mode both in simulation and in the real world, so for integrating inertial data, the approach used was filtering the Visual Odometry from ORBSLAM and the IMU data using an extended kalman filter (EKF). We used the ROS2 package robot_localization that contains an already implemented EKF. The filtered odometry was less noisy compared to the ORB-SLAM3 odometry, and the scale error, although still important on the z axis, was reduced. In addition to this, using an EKF allowed us to have estimation not only of the position and orientation, but also of the linear and angular velocity, and of the linear acceleration.

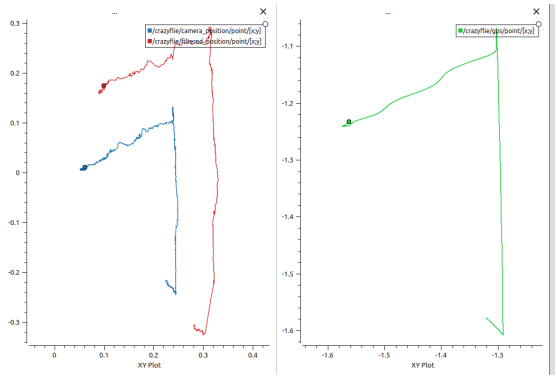


Fig. 3: Comparison between the RGB only odometry (blue), the EKF filtered odometry (red) and the ground truth (green) on the xy plane. The scale error is strongly reduced

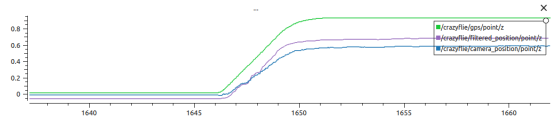


Fig. 4: Comparison between the rgb only odometry (blue), the ekf filtered odometry (purple) and the ground truth (green) on the z. The scale error is slightly reduced.

IV. 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 [21]. For a detailed explanation of differential flatness refer to appendix B. More formally, the flat outputs are defined as

$$\sigma = [x, y, z, \psi]^T \tag{1}$$

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:

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

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 [31] to generate smooth trajectories $\sigma(t)$

in the flat output space in an fast and efficient manner using [31] and their open-source implementation[4].

A. Minimum Snap/ Jerk Trajectory Generation

This section closely follows the notation of [31] and formalises the used method. Since the spatial dimensions can be decoupled for minimum snap/ jerk polynomial trajectory generation [6] 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 $(p_1, \dots, p_M)$ where every spline $p_i, i \in [M]$ is a N-degree polynomial [31]:

$$p_i : [0, T_i] \to \mathbb{R} : t \mapsto c_i^T \beta(t) \tag{3}$$

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

$$p : [0, \tau_M] \to \mathbb{R} : t \mapsto p_i(t - \tau_{i-1}), \quad t \in [\tau_{i-1}, \tau i] \tag{4}$$

for the entire spline where $\tau_i := \sum_{j=1}^{i} T_j$, and $c = (c_1^T, \dots, c_M^T)$ as the full coefficient vector of p. Define $s = 3$ for jerk, and $s = 4$ for snap minimization. We set $N = 2s - 1$ as optimal degree [30]. The trajectory minimisation problem can now be formulated as follows

$$\min_{c,T} \int_0^{\tau_M} p^{(s)}(t)^2 dt \tag{5}$$

$$s.t. \quad p^{(j)}(0) = d_{0,j}, \quad 0 \le j < s \qquad \text{(Start derivatives)}$$

$$p^{(j)}(\tau_M) = d_{M,j}, \quad 0 \le j < s \qquad \text{(End derivatives)}$$

$$p(\tau_i) = q_i, \quad 0 \le i < M \qquad \text{(Position constraints)}$$

where $d_0 := (d_{0,0}, \dots, d_{0,s-1})^T$ and $d_M := (d_{M,0}, \dots, d_{M,s-1})^T$ are initial and final derivatives, respectively, and $q := (q_0, \dots, q_{M-1})$ 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. (5). While there exist a closed-form solution to eq. (5) [6], its computation is inefficient due to the numerical computation of a matrix inverse [31]. We adopt the linear complexity algorithm proposed by Wang et al. [31] which does not necessitate numerically computing the matrix inverse.

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

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

B. Random Trajectory Generation

To train a generalizing RL policy (see section VI), 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 previously defined minimum snap/ jerk trajectory generation algorithm and feed it with random target/ control waypoints q and uniform timestamps T. Algorithm 1 illustrates our algorithm in pseudo-code, 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 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 appendix A.

V. Model Predictive Control

MPC uses mathematical optimization to control the state of the drone. Given an initial state and a goal state, it uses a dynamics model to predict its state based on chosen inputs. This forward-looking capability allows for the derivation of the optimal control inputs to achieve the optimal state minimizing a user configured sum of distances, the objective function.
The model used for the prediction is the following system of differential equations (specific values for I_x, I_y, I_z and m are given in the Appendix) [12] [5]:

$$\begin{aligned}
\ddot{x} &= (\cos\phi \cdot \sin\theta \cdot \cos\psi + \sin\phi \cdot \sin\psi) \cdot \frac{u_1}{m} \\
\ddot{y} &= (\cos\phi \cdot \sin\theta \cdot \sin\psi - \sin\phi \cdot \cos\psi) \cdot \frac{u_1}{m} \\
\ddot{z} &= (\cos\phi \cdot \cos\theta) \cdot \frac{u_1}{m} - g
\end{aligned} \tag{6}$$

$$\begin{aligned}
\ddot{\phi} &= \dot{\theta}\dot{\psi}\left(\frac{I_y - I_z}{I_x}\right) + \frac{u_2}{I_x} \\
\ddot{\theta} &= \dot{\phi}\dot{\psi}\left(\frac{I_z - I_x}{I_y}\right) + \frac{u_3}{I_y} \\
\ddot{\psi} &= \dot{\theta}\dot{\phi}\left(\frac{I_x - I_y}{I_z}\right) + \frac{u_4}{I_z}
\end{aligned} \tag{7}$$

with [17]

$$\begin{aligned}
u_1 &= F_{thrust} = (M_1 + M_2 + M_3 + M_4) \\
u_2 &= \tau_\phi = (M_1 - M_3) * l \\
u_3 &= \tau_\theta = (M_2 - M_4) * l \\
u_4 &= \tau_\psi = (-M_1 + M_2 - M_3 + M_4) * C.
\end{aligned} \tag{8}$$

Defining $x(t)$ as the the state of the drone $\underline{x}(t) = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}]^T$ and $\underline{u}(t)$ as the control inputs with $\underline{u}(t) = [u_1, u_2, u_3, u_4]^T$ we get with eq. (6) and eq. (7) [12]:

$$\dot{\underline{x}}(t) = f(\underline{x}(t), \underline{u}(t)) \tag{9}$$

which we discretize and integrate with the Runge-Kutta fourth-order method to:

$$\underline{x}(t+1) = F(\underline{x}(t), \underline{u}(t)). \tag{10}$$

The objective function, aimed at minimizing to derive the optimal control inputs, is J_N. The optimization problem is stated with the following constraints:

$$\begin{aligned}
\min_{\mathbf{u}} \quad & J_N(\mathbf{x}, \mathbf{u}) = \sum_{k=0}^{N} \left(\|\mathbf{x}(k) - \mathbf{x}_{ref}\|_Q^2 + \|\mathbf{u}(k) - \mathbf{u}_{ref}\|_R^2 \right) \\
\text{s.t.} \quad & \mathbf{x}(k+1) = F(\mathbf{x}(k), \mathbf{u}(k)) \\
& \mathbf{x}(0) = \mathbf{x}_0 \\
& u_{min} \leq u_i \leq u_{max}.
\end{aligned} \tag{11}$$

Stable and effective control results (Appendix subsection G) were achieved in PyBullet simulation environment with the CasADi framework [3] with the following constraints on control inputs and weighting matrices Q and R inspired by [12]:

$$\begin{aligned}
0 \leq & F_{thrust} \leq 35 \\
-1.257 \leq & \tau_\phi \leq 1.257 \\
-1.257 \leq & \tau_\theta \leq 1.257 \\
-0.2145 \leq & \tau_\psi \leq 0.2145
\end{aligned} \tag{12}$$

$$\begin{aligned}
Q &= \text{diag}(1, 1, 1, 0.3, 0.3, 0.2, 0, 0, 0, 0, 0, 0) \\
R &= \text{diag}(0.15, 0.15, 0.15, 0.4).
\end{aligned} \tag{13}$$

The weighting matrices proved to be the most effective among a variety of ideas tested to improve performance through the tuning of weighting matrices. Among these ideas were the complete removal of constraints on the orientation and the control inputs and the adaptive k-weighting matrices approach. Details and variants of these approaches are detailed in the appendix.

The resulting control inputs were converted to PWM input signals sent to the drone in PyBullet using the mappings (Input Command (PWM) $\rightarrow$ Thrust) and (Thrust $\rightarrow$ Torque) detailed in the appendix and identified in this work [15]. For our purposes, (Thrust $\rightarrow$ Torque) was inverted and approximated to (Torque $\rightarrow$ Thrust):

$$f_i = 1676.57185318 \cdot \tau_i \tag{14}$$

and the quadratic mapping (Input Command (PWM) $\rightarrow$ Thrust) was inverted to (Thrust $\rightarrow$ Input Command (PWM)):

$$\text{pwm}_i = \begin{cases} -24236.9 + 1.57508 \times 10^{-11} \cdot \\ \sqrt{1.89215 \times 10^{32} f_i + 2.26 \times 10^{30}}, & f_i \geq 0, \\ \\ -1 \times (-24236.9 + 1.57508 \times 10^{-11} \cdot \\ \sqrt{1.89215 \times 10^{32} (-f_i) + 2.26 \times 10^{30}}), & f_i < 0. \end{cases} \tag{15}$$

A mapping (Torque $\rightarrow$ Input Command (PWM)) was then created by concatenating (Torque $\rightarrow$ Thrust) and (Thrust $\rightarrow$ Input Command (PWM)). This mapping was then used to map the output of the CasADi MPC controller to the input of the PyBullet simulation.

VI. Reinforcement Learning

As the field of quadrotors advances, the requirements for what constitutes a 'successful' quadrotor are continually being increased. Current control methodologies, such as MPC, have demonstrated proficiency in managing the dynamics of these systems under various conditions. However, as the environments become increasingly complex, marked by the need to navigate cluttered or dynamically changing spaces, these traditional control methods begin to face limitations. Specifically, the challenge intensifies when there is a need to process high-dimensional data and to exhibit high levels of adaptability in real-time. This is where RL comes as a

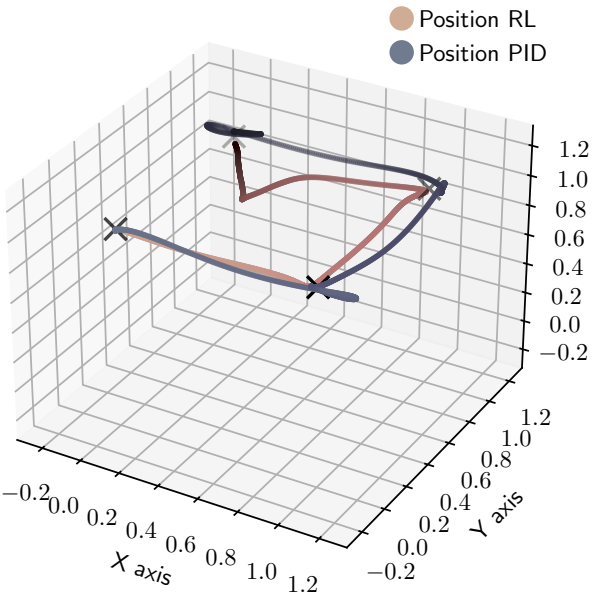


Fig. 5: Simple position controller RL agent on a 4-waypoint following task, compared against a position PID controller.

powerful tool to address these problems. Motivated by the potential of RL in many areas of robotics and beyond, we decided to adapt these methods for the problem of trajectory following and quadrotor landing.

A. Position Controller

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:

$$r(\boldsymbol{p}, \boldsymbol{t}) = max(0, 2 - ||\boldsymbol{p} - \boldsymbol{t}||^4) + \mathbf{100}_{\{\|p-t\| \leq r\}}, \qquad (16)$$

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 5 shows our obtained policy.

B. Trajectory Follower

In the area of drone racing, Penicka et al. [25] 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 [25]. 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 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. [16] 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:

$$r(t) = k_p r_p(t) + k_s r_s(t) + k_{wp} r_{wp} + r_T, \qquad (17)$$

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 [26]. Instead, as a supplement, we provide a fig. 6 illustrating these terms. The reward from eq. (17) 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 [25] 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 [25] for more details. The final expression for reward signal is

$$r(t) = \alpha k_p r_p(t) + \alpha k_s r_s(t) + k_{wp} r_{wp} + r_T. \qquad (18)$$

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

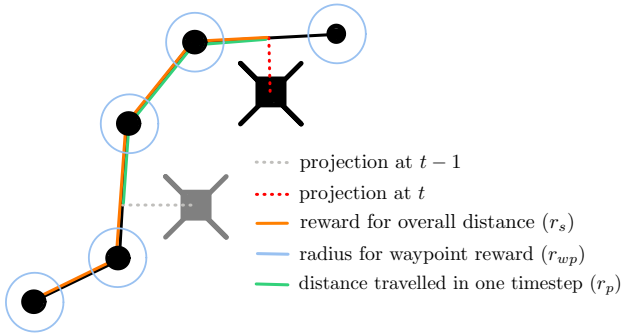


Fig. 6: Illustration of different components of the reward signal. The grey quadrotor represents the agent at timestep $t-1$, the black quadrotor represents the agent at timestep t.

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.

C. 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.

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**

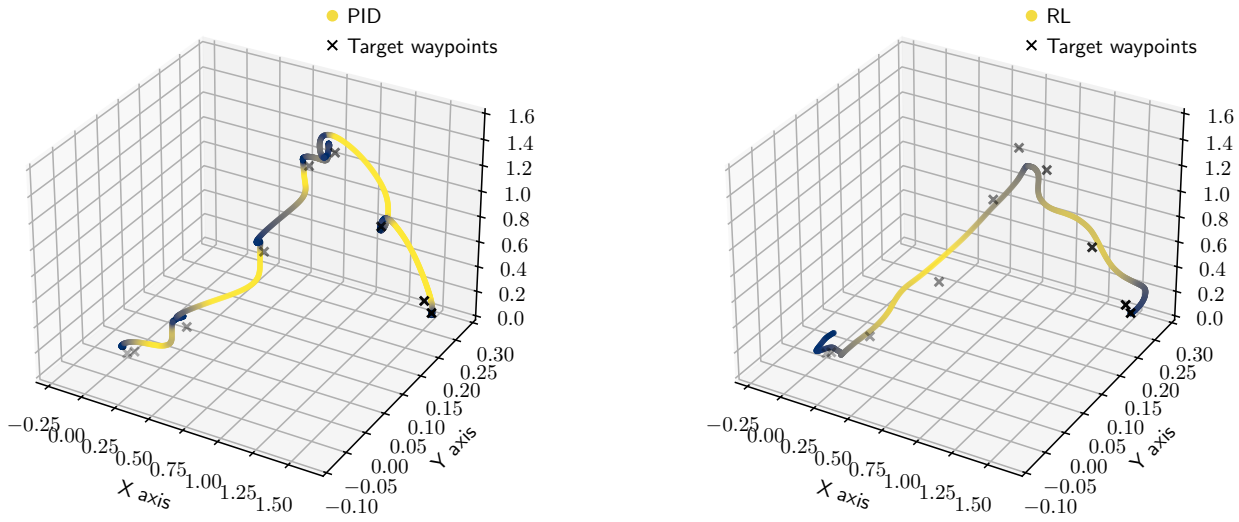
TABLE I: Comparison of our reinforcement learning (RL) policy against a position PID controller. Averages and comparable runs are shown.

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 7a 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. 7b, 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. We refer to appendix D for a more detailed description.

VII. DISCUSSION AND FUTURE WORK

In the RL part of the project we got our hands on the drone simulation environment (pybullet), integrated our flexible trajectory generation mechanism into a training pipeline,



(a) PID. (b) RL.

Fig. 7: Trajectories are color-coded according to velocity. Blue corresponds to slow, whereas yellow corresponds to fast.

successfully implemented an influential paper from drone racing [25] and managed to run our experiments in cloud, which successfully fulfilled authors' expectations from this practical course. Our policy produces smooth and natural control, as seen in fig. 7b. 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 weakness is that the agent is unable to follow trajectories that intersect themselves. This is due to the fact, that during computation of the closest point on the trajectory, we consider all line segments of a given trajectory. We tried to resolve it by computing projection only on the 'surrounding' line segments, but our definition of what 'surrounding segments' are was introducing problems in learning, leading to an undesired behavior. At the time of writing this chapter, we think we have a remedy for this issue, but the time budget that we have left is not sufficient to apply it. We therefore leave this as a potential for future work. Another area for future work would be to make our logging and evaluation code more suitable for large scale experiments and evaluation, as well as switching to a more parallelization friendly simulator. For MPC, a real-time capable implementation should be considered, especially if future work aims at real-world hardware implementation. The given CasADi Python MPC controller could be rewritten in C++. Another approach would be the usage of the TinyMPC code library (further details are delineated in the appendix). A different idea to potentially improve the MPC controller would be to use RL to find appropriate weighting matrices and thresholds for control inputs.

VIII. Conclusion

In this project, we successfully implemented a ROS2 software that works in a Webots simulation environment. For the real-world application, we successfully implemented the interfaces to the UAV, i.e. a working video and IMU data stream, and sending control commands. The state estimation was successfully implement using ORBSLAM3 and integrated into our ROS2 software. Due to limitations of the setup, i.e. unstable video stream and an improvised hardware setup, the second half of the project focused on working in the simulation. Due to limitations of Webots when used for RL, we moved to the `gym-pybullet-drones` simulation environment. There, we successfully implemented a MPC, and trained a generalizable RL-based control policy that outputs stable and smooth control. To train the RL policy, we leveraged randomly generated feasible trajectories using minimum-snap polynomial trajectory generation. We evaluate our trained policy on two test sets, one for landing and one for long flight, and show that our policy improves upon a baseline position PID controller.

References

[1] Anoushka Alavilli, Khai Nguyen, Sam Schoedel, Brian Plancher, and Zachary Manchester. Tinympc: Model-predictive control on resource-constrained microcontrollers. *arXiv preprint arXiv:2310.16985*, 2023.

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

[3] Joel AE Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. Casadi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11:1–36, 2019.

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

[5] Samir Bouabdallah. Design and control of quadrotors with application to autonomous flying. Technical report, Epfl, 2007.

[6] 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.

[7] 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.

[8] 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.

[9] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.

[10] 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.

[11] 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.

[12] Mohamed Elhesasy, Tarek N. Dief, Mohammed Atallah, Mohamed Okasha, Mohamed M. Kamra, Shigeo Yoshida, and Mostafa A. Rushdi. Non-Linear Model Predictive Control Using CasADi Package for Trajectory Tracking of Quadrotor. *Energies*, 16(5):2143, January 2023. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

[13] Espressif. Esp32-camera. https://github.com/espressif/esp32-camera, 2024. [Online; accessed 13-Feb-2024].

[14] Dario Floreano and Robert J Wood. Science, technology and the future of small autonomous drones. *nature*, 521(7553):460–466, 2015.

[15] Julian Förster. System identification of the crazyflie 2.0 nano quadrocopter. B.S. thesis, ETH Zurich, 2015.

[16] Elia Kaufmann, Leonard Bauersfeld, and Davide Scaramuzza. A benchmark comparison of learned control policies for agile quadrotor flight, 2022.

[17] Jinho Kim, S. Andrew Gadsden, and Stephen A. Wilkerson. A Comprehensive Survey of Control Strategies for Autonomous Quadrotors. *Canadian Journal of Electrical and Computer Engineering*, 43(1):3–16, 2020. Conference Name: Canadian Journal of Electrical and Computer Engineering.

[18] 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.

[19] Steven Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. Impact of ros 2 node composition in robotic systems. *IEEE Robotics and Autonomous Letters (RA-L)*, 2023.

[20] Gregory S McNeal. Drones and the future of aerial surveillance. *Geo. Wash. L. Rev.*, 84:354, 2016.

[21] 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.

[22] Olivier Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[23] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE international conference on robotics and automation*, pages 3400–3407. IEEE, 2011.

[24] 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.

[25] 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.

[26] 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.

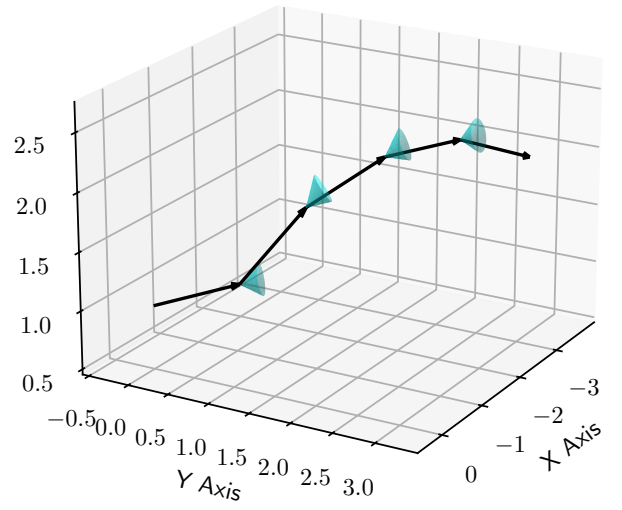
[27] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[28] Sony. Dualshock ps4 controller. https://www.playstation.com/de-de/accessories/dualshock-4-wireless-controller/, 2024. [Online; accessed 13-Feb-2024].

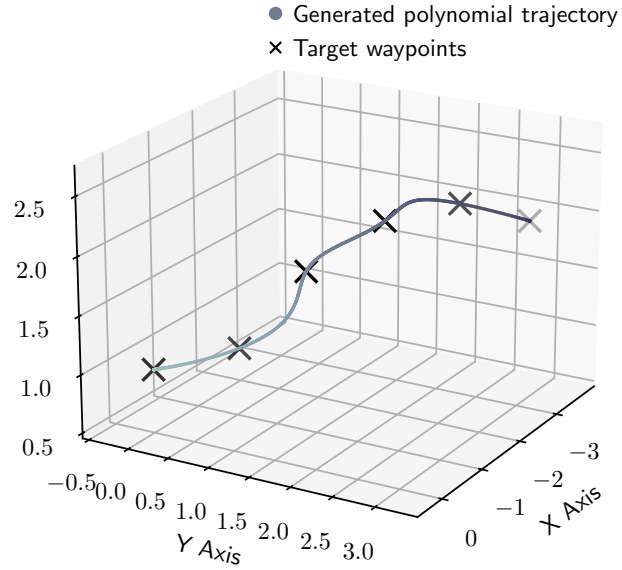
[29] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023.

[30] EI Verriest and FL Lewis. On the linear quadratic minimum-time problem. *IEEE transactions on automatic control*, 36(7):859–863, 1991.

[31] 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.



(a) Random control point generation.



(b) Minimum snap trajectory generation.

Fig. 8: 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 [31].

APPENDIX

A. Random Trajectory Generation Example

In section IV-B, we described the algorithm on how we generate randomized trajectory for training our RL policy. In fig. 8, a visual illustration of this algorithm can be found by providing an example generated trajectory.

B. Differential Flatness

Differential flatness has been validated by [21] and, since then, led to a series of follow-up works [6], [7], [10], [11]. Recall that the trajectory is now defined in the space of flat outputs:

$$\sigma(t) : [t_0, t_m] \rightarrow \mathbb{R}^3 \times SO(2) \tag{19}$$

see eq. (2). 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 [21], which builds the foundation of the literature for polynomial trajectory generation. For the full derivation, refer to [21].

C. Improving the Video Stream with Packet Injection

For improving the stability and latency of the video stream, The approached solution based on open-source code[5] involves sending the video data via packet injection because it allows for transmitting data as it is received via direct memory access (DMA), thus omits overhead due to buffering and assembling complete frames before transmission. The sent packets then can be read via radio frequency monitor (RFMON) (also: monitor mode). Since Ubuntu does not support RFMON, a separate device captures the packets and forwards them to the Ubuntu device via ethernet in an 1-to-1 mapping. While the solution is conceptually sound, it requires a deep dive into low-level communication primitives written in C and a detailed understanding of the structure of the headers of WiFi packets in the Data Link Layer (Layer 2), including nuanced details such as the interpretation of radiotap headers. A deep dive into the code revealed that this task is out of the scope of this project.

D. 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,

[5]https://github.com/jeanlemotan/esp32-cam-fpv

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.

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.

E. PID Controller

We implemented a position PID controler built on top of an attitude PID controler. In total there are 6 PID controllers, 3 corresponding to x,y and z position for the position PID, and 3 corresponding to roll, pitch and yaw for the attitude PID controller. The x and y PIDs output a target pitch and roll, used as input by the pitch and roll PIDs. All the parameters of all the PIDs were manually tuned. The architecture of the PID cascade architecture can be found in fig. 10.

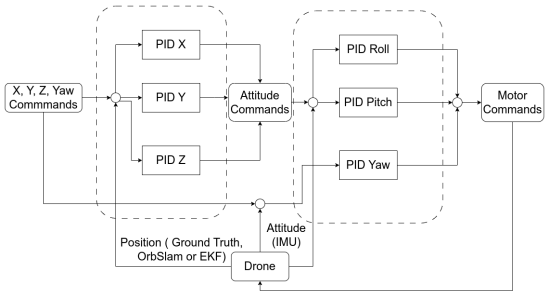


Fig. 9: Diagram of the PIDs cascade architecture.

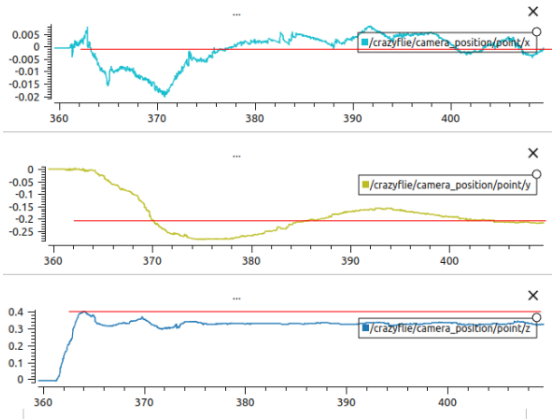


Fig. 10: Result of the PID controller to the target (0.0 -0.2 0.4) on the x, y and z axis. The robot position is estimated using Orbslam.

F. System identification for Model Predictive Control

a) Inertia and mass of Crazyflie:

$$\begin{aligned} Ix &= 0.0000166 \quad kg \cdot m^2 \\ Iy &= 0.0000167 \quad kg \cdot m^2 \\ Iz &= 0.00000293 \quad kg \cdot m^2 \\ m &= 29.0\ g \end{aligned} \tag{20}$$

[15]

b) Mappings:

(Input Command (PWM) $\rightarrow$ Thrust)

$$\begin{aligned} f_i =& 2.130295 \cdot 10^{-11} \cdot \text{pwm}_i^2 \\ &+ 1.032633 \cdot 10^{-6} \cdot \text{pwm}_i + 5.484560 \cdot 10^{-4} \end{aligned} \tag{21}$$

[15]

(Thrust $\rightarrow$ Torque)

$$\tau_i = 0.005964552 \cdot f_i + 1.563383 \cdot 10^{-5} \tag{22}$$

[15]

G. MPC-tuning: Weight matrices Q and R

During implementation

$$\begin{aligned} Q &= \text{diag}(1, 1, 1, 0.6, 0.6, 1, 0, 0, 0, 0, 0, 0) \\ R &= \text{diag}(0.3, 0.3, 0.3, 0.8) \end{aligned} \tag{23}$$

were weighting matrices adopted from [12]. The resulting control was effective but visual inspection in the PyBullet simulation revealed that it was a bit too rigid and slow.

Thus the following weighting matrix configurations were considered, which were supposed to loosen up constraints on the orientation and the control inputs:

$$\begin{aligned} Q &= \text{diag}(1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0) \\ R &= \text{diag}(0, 0, 0, 0) \end{aligned} \tag{24}$$

$$\begin{aligned} Q &= \text{diag}(1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0) \\ R &= \text{diag}(0.3, 0.3, 0.3, 0.8) \end{aligned} \tag{25}$$

Both configurations led to an unstable flying behavior. We conclude that the term $\|\mathbf{u}(k) - \mathbf{u}_{ref}\|_R^2$ and the weights on the orientation have thus a necessary stabilizing control effect on the drone.

Furthermore, an adaptive k-weighting matrices approach was tested:

$$\begin{aligned} Q_k &= \frac{k}{N_{\text{horiz}}} \cdot Q, \\ R_k &= \frac{k}{N_{\text{horiz}}} \cdot R \end{aligned} \tag{26}$$

where $k \in \{0, 1, 2, ..., N_{\text{horiz}} - 1\}$.

The idea was to gradually apply the weighting the further the state is in the prediction horizon and have more freedom of action in the beginning. Multiple variants of this idea were also tested such as the following:

$$\begin{aligned} Q_k &= \left(1 - \frac{k}{N_{\text{horiz}}}\right) \cdot Q, \\ R_k &= \left(1 - \frac{k}{N_{\text{horiz}}}\right) \cdot R, \end{aligned} \tag{27}$$

where $k \in \{0, 1, 2, ..., N_{\text{horiz}} - 1\}$ is the incremented variable. The approach here was to gradually discount the weighting the further the state is in the prediction horizon. The observed results were mixed. While the approaches occasionally enhanced performance on certain tracks, it also occasionally led to less precise control, causing the drone to deviate from its intended trajectory. Further exploration and testing is necessary to fully realize the potential of this concept.

More details can be found in the commit messages detailing every test result of various weight matrix configurations[6]

In the end, the best performing weighting matrix configuration is:

$$\begin{aligned} Q &= \text{diag}(1, 1, 1, 0.3, 0.3, 0.2, 0, 0, 0, 0, 0, 0) \\ R &= \text{diag}(0.15, 0.15, 0.15, 0.4) \end{aligned} \tag{28}$$

[6]https://github.com/danielbinschmid/RL-pybullets-cf/compare/main...test_suite_MPC

	PID	MPC
Success Rate	95%	100%
Avg Deviation (cm)	6.64	9.995
Avg Max Deviation (cm)	18.06	23.87
Average time (s)	4.54	4.12
Simulation Speed	0.4x	0.04x (CasADi Python)

TABLE IV: Performance of PID vs. MPC in test suite of 20 tracks in PyBullet.

H. Comparison between PID vs. MPC

A comparison between the PID controller and the MPC controller with the above weighting matrices was conducted. During testing a unresolved bug was discovered: For some reason the drone took off erratically shortly before the origin position of the test trajectory $(0, 0, 0)^T$ if certain if statements for trajectory waypoint switching and trajectory termination were set. The if statements were a close distance condition and a velocity condition for the termination, where the norm of the drone's velocity had to be less than 0.05.
A remedy to this problem was found: The trajectory waypoint switched to the next if the distance of the drone to the current waypoint was only below 20 cm without the velocity constraint. The trajectory test also terminated with the same condition. With this trajectory termination condition the results in Table IV have to be interpreted as an evaluation of general trajectory following without a landing or stopping procedure. Especially, the measured "Average time (s)" have to interpreted as the time taken to reach within 20 cm of the last position. This remedy was chosen due to time reason.

I. TinyMPC

TinyMPC is a real-time capable MPC solver with a low memory usage well matched for micro-controllers typically used on the Crazyflie. [1] Their approach leverages the structure of the discrete Riccati equation of the MPC problem for efficiency and is based on the alternating direction method of multipliers (ADMM). [1]

A C++ TinyMPC ROS2 node using the Webots simulator was implemented[7] during this project and was considered as a solution until bugs were observed. The major bug was a negative yaw spinning behavior and the minor bug was that a slight perturbation in values of the initial state, led to - if the initial state was a zero vector - a completely different outcome. It is unclear where the root of these bugs come from. The maintainers were contacted on GitHub[8], but the problem remains unsolved. A contact was given by one of the maintainers, stating that a person they knew, implemented TinyMPC successfully in PyBullet. After a private E-mail conversation, we found out that the implementation did not work (drone just flies away)[9]. Further inquiries did not receive a response at the time of writing this report.

[7]https://github.com/AlboAlby00/CrazyflieControllers/tree/TinyMPC
[8]https://github.com/orgs/TinyMPC/discussions/14
[9]https://github.com/ucb-bar/Accelerated-TinyMPC/blob/extended_functionality/python_wrapper_for_tracking.py