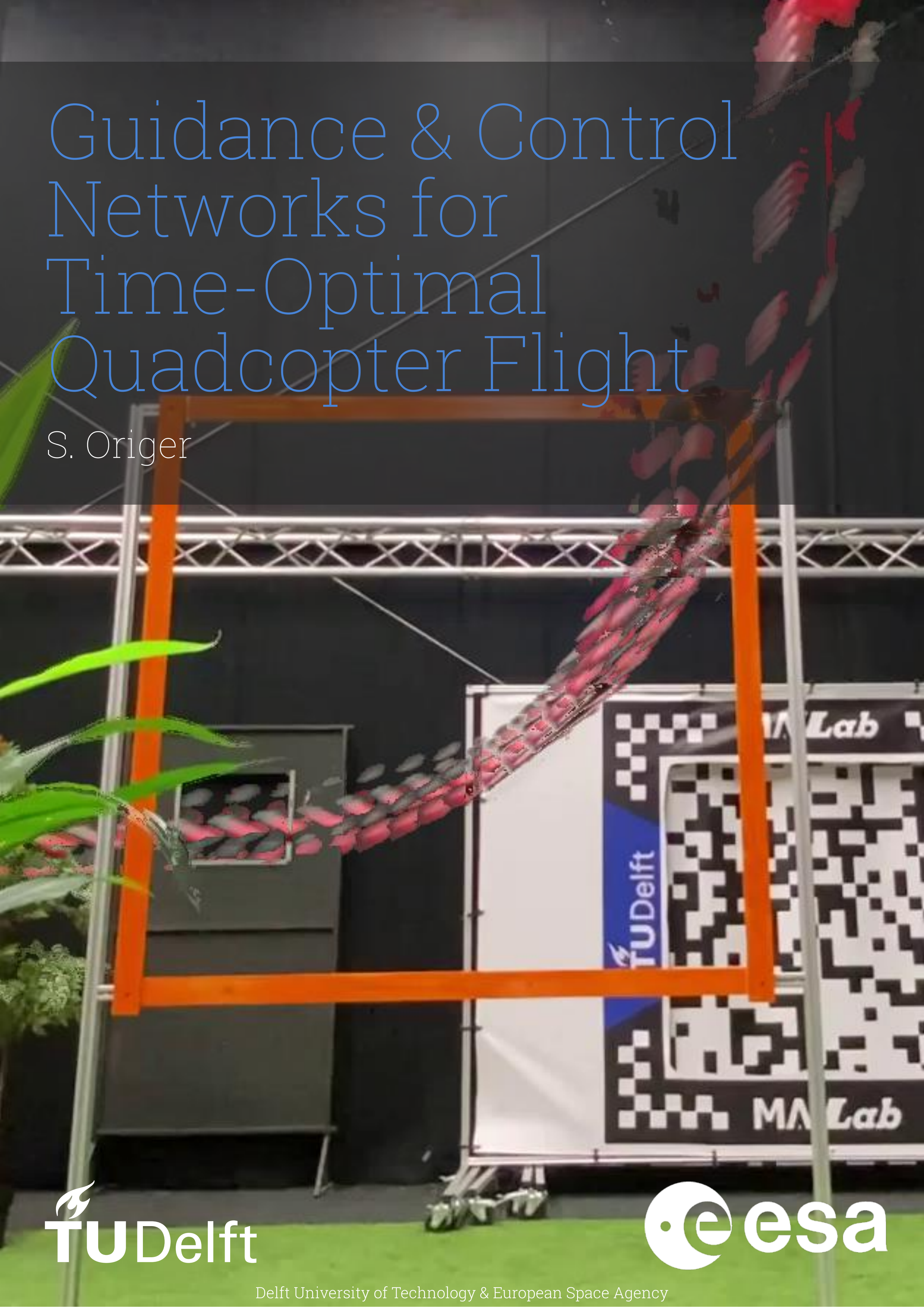
Guidance & Control Networks for Time-Optimal Quadcopter Flight

S. Origer



Delft University of Technology & European Space Agency

Guidance & Control Networks for Time-Optimal Quadcopter Flight

by

S. Origer

Student Name	Student Number
Sebastien Origer	4662792

Supervisors:	Christophe De Wagter, Guido C.H.E. de Croon, Dario Izzo, Robin Ferede
Thesis Duration:	February, 2022 - April, 2023
Faculty:	Faculty of Aerospace Engineering, Delft University of Technology
In collaboration with:	Advanced Concepts Team, European Space Agency
Cover:	Parrot Bebop 1 flight in Cyberzoo (TU Delft)

TU Delft

Preface

I am extremely grateful to have had the chance to work for over a year with Robin Ferede, Christophe De Wagter, Guido C.H.E. de Croon, Dario Izzo, Emmanuel Blazquez and Alexander Hadjiivanov. I consider you all to not only be great scientists but also great teachers. Your words of encouragement mean a lot to me and I truly hope to be collaborating with you in the future. I also want to thank my friends from Delft and Luxembourg for always letting me share my joy regarding this project and giving me helpful advice along the way. I am really lucky to have met you all and I could not wish for better friends. Finally, I want to thank my mum, my dad, my brother and my sister for their unconditional support throughout the last five and a half years in Delft. Without you, none of this would have been possible.

S. Origer
Delft, April 2023

Abstract

Reaching fast and autonomous flight requires computationally efficient and robust algorithms. To this end, we train Guidance & Control Networks to approximate optimal control policies ranging from energy-optimal to time-optimal flight. We show that the policies become more difficult to learn the closer we get to the time-optimal 'bang-bang' control profile. We also assess the importance of knowing the maximum angular rotor velocity of the quadcopter and show that over- or underestimating this limit leads to less robust flight. We propose an algorithm to identify the current maximum angular rotor velocity onboard and a network that adapts its policy based on the identified limit. Finally, we extend previous work on Guidance & Control Networks by learning to take consecutive waypoints into account. We fly a 4×3m track in similar lap times as the differential-flatness-based minimum snap benchmark controller while benefiting from the flexibility that Guidance & Control Networks offer.

Contents

Preface **i**

Abstract **ii**

1 Introduction **1**

I Scientific Paper **3**

II Literature review **19**

III Addition results **56**

References **66**

1 Introduction

Micro air vehicles (MAVs) are very versatile robots with clear applications such as search and rescue missions, entertainment, cinematography, delivery and inspection. Their size, agility, speed, vertical take-off and landing capabilities open up a world of possibilities yet to be explored. In addition, their low cost and the intrinsic challenges of autonomous flight make them the ideal platform to push the frontiers of robotics research.

Making autonomous and time-optimal flight a reality is relevant as the demand for drones that can perform tasks autonomously is on the rise. Flying time-optimally is important as the success of some applications, such as search and rescue missions, hinges on how quickly the drone can reach its destination. In addition, applications that require long flight ranges, such as inspection of offshore wind turbines, also stand to benefit from time-optimal control solutions, as the optimal speed for range is generally relatively fast for multicopters [1]. This is especially important for drones that are not equipped with a fixed wing, such as quadcopters, as these suffer from limited flight ranges. Autonomous and time-optimal flight has already been the subject of many studies [2, 10, 14, 13, 9, 12, 7, 4] and drone racing competitions such as the AlphaPilot challenge [2] are being organized to further promote research in this field. Since aerodynamics effects become more significant for time-optimal flight [15] and drones suffer from limited onboard computing power, one of the challenges to overcome is to create robust and computationally efficient control algorithms.

Traditionally, the problem of autonomous flight is broken down into three major steps: perception (state estimation), planning (trajectory generation) and control (trajectory tracking). Some of these steps can be fused together, in fact, one often distinguishes between two types of control strategies: trajectory tracking methods and trajectory optimization methods. The former first generates the trajectory and then tracks it with a controller, whereas the latter combines both steps together, i.e. computes the control commands directly from the states. While state-of-the-art trajectory tracking methods, such as differential-flatness-based-control (DFBC) [11] or model predictive contouring control (MPCC) [14] achieve high speeds, it is usually the first step (the trajectory generation) that is too computationally expensive to be solved onboard of drones [14], unless simplified models are used, such as point-mass models. This is problematic as disturbances are bound to make the quadcopter deviate from the pre-computed trajectory.

Recent advances in machine learning show promising results in the three major steps of autonomous flight [2, 7, 8, 3]. Given a well-posed problem, a large enough training dataset, suitable network architecture and enough computational power, artificial neural nets can today learn to approximate any function up to a certain accuracy. The main machine learning paradigms for guidance and control tasks are reinforcement learning (RL) [7] and imitation learning [3]. RL offers a framework where a network can learn to deal with uncertainty by adding noise to the environment, which is particularly useful when parts of the dynamic system remain unmodelled. However, it is also possible to deal with unmodelled effects using imitation learning [3]. In addition, previous work in the context of interplanetary transfers [16, 6, 5] and quadcopters [3, 8] has shown that a network can directly learn the optimal state feedback from a large dataset of optimal trajectories. This has led to the term Guidance & Control Networks (G&CNETs). G&CNETs offer a direct mapping from states to raw control commands, they can be inferred at a low computational cost and they are very flexible since there is no need to recompute optimal trajectories.

In this work, we improve past work on G&CNETs [3] to increase the quadcopter's flight speed, leading to four main contributions. We make the step from energy-optimal to time-optimal control and provide the loss values when training G&CNETs to approximate the corresponding optimal control policies. We introduce an adaptive scheme that can estimate the maximum angular velocity of the propellers and use it to adjust the commands to remain time-optimal. We develop a training method that allows for the network to output time-optimal raw control commands taking a horizon into account of the next two waypoints. We demonstrate that the two-waypoints network can deal with dynamic waypoint locations during the flight.

This thesis document is structured in three parts: Part.I contains the scientific paper, Part.II the literature review and Part.III additional results which might come in handy to anyone continuing this work.

Part I

Scientific Paper

Guidance & Control Networks for Time-Optimal Quadcopter Flight

Sebastien Origer[1,†], Christophe De Wagter[1,‡], Robin Ferede[1,‡], Guido C.H.E. de Croon[1,‡], Dario Izzo[2,‡]

Abstract— **Reaching fast and autonomous flight requires computationally efficient and robust algorithms. To this end, we train Guidance & Control Networks to approximate optimal control policies ranging from energy-optimal to time-optimal flight. We show that the policies become more difficult to learn the closer we get to the time-optimal 'bang-bang' control profile. We also assess the importance of knowing the maximum angular rotor velocity of the quadcopter and show that over- or underestimating this limit leads to less robust flight. We propose an algorithm to identify the current maximum angular rotor velocity onboard and a network that adapts its policy based on the identified limit. Finally, we extend previous work on Guidance & Control Networks by learning to take consecutive waypoints into account. We fly a $4 \times 3\text{m}$ track in similar lap times as the differential-flatness-based minimum snap benchmark controller while benefiting from the flexibility that Guidance & Control Networks offer.**

Index Terms— **G&CNET, optimal control, imitation learning, end-to-end, time-optimal**

Video: `https://youtu.be/FrwpODT0HKQ`

I. Introduction

MICRO air vehicles (MAVs) are very versatile robots with clear applications such as search and rescue missions, entertainment, cinematography, delivery and inspection. Their size, agility, speed, vertical take-off and landing capabilities open up a world of possibilities yet to be explored. In addition, their low cost and the intrinsic challenges of autonomous flight make them the ideal platform to push the frontiers of robotics research.

Making autonomous and time-optimal flight a reality is relevant as the demand for drones that can perform tasks autonomously is on the rise. Flying time optimally is important as the success of some applications, such as search and rescue missions, hinges on how quickly the drone can reach its destination. In addition, applications that require long flight ranges, such as inspection of offshore wind turbines, also stand to benefit from time-optimal control solutions, as the optimal speed for range is generally relatively fast for multicopters [1]. This is especially important for drones that are not equipped with a fixed wing, such as quadcopters, as these suffer from limited flight ranges. Autonomous and time-optimal flight has already been the subject of many

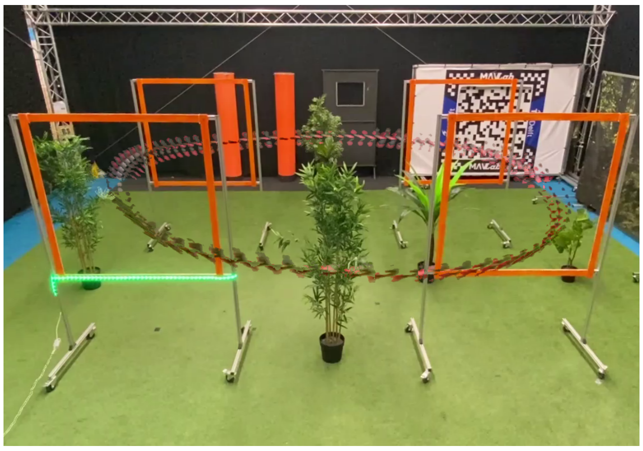


Fig. 1. Flight path of the fastest lap using the Parrot Bebop 1 and a Guidance & Control Network which learned to take two upcoming waypoints into account to compute the optimal control inputs. The track is a $4 \times 3\text{m}$ rectangle, and the four waypoints are positioned at the center of each orange gate.

studies [2]–[9] and drone racing competitions such as the AlphaPilot challenge [2] are being organized to further promote research in this field. Since aerodynamics effects become more significant for time-optimal flight [10] and drones suffer from limited onboard computing power, one of the challenges to overcome is to create robust and computationally efficient control algorithms.

Traditionally, the problem of autonomous flight is broken down into three major steps: perception (state estimation), planning (trajectory generation) and control (trajectory tracking). Some of these steps can be fused together, in fact, one often distinguishes between two types of control strategies: trajectory tracking methods and trajectory optimization methods. The former first generates the trajectory and then tracks it with a controller, whereas the latter combines both steps together, i.e. computes the control commands directly from the states. While state-of-the-art trajectory tracking methods, such as differential-flatness-based-control (DFBC) [11] or model predictive contouring control (MPCC) [4] achieve high speeds, it is usually the first step (the trajectory generation) that is too computationally expensive to be solved onboard of drones [4], unless simplified models are used, such as point-mass models. This is problematic as disturbances are bound to make the quadcopter deviate from the pre-computed trajectory.

Recent advances in machine learning show promising results in the three major steps of autonomous flight [2], [8], [12], [13]. Given a well-posed problem, a large enough training dataset, suitable network architecture and enough

*This work was supported by ESA

†Student, ‡Supervisors

[1]Micro Air Vehicle Lab of the Faculty of Aerospace Engineering, Delft University of Technology, 2629 HS Delft, The Netherlands `Sebastien.Origer@outlook.com`, `C.deWagter@tudelft.nl`, `R.Ferede@tudelft.nl`, `G.C.H.E.deCroon@tudelft.nl`

[2]Advanced Concepts Team, European Space Agency, Keplerlaan 1, 2201 AZ, Noordwijk, The Netherlands. `Dario.Izzo@esa.int`

computational power, artificial neural nets can today learn to approximate any function up to a certain accuracy. The main machine learning paradigms for guidance and control tasks are reinforcement learning (RL) [8] and imitation learning [13]. RL offers a framework where a network can learn to deal with uncertainty by adding noise to the environment, which is particularly useful when parts of the dynamic system remain unmodelled. However, it is also possible to deal with unmodelled effects using imitation learning [13]. In addition, previous work in the context of interplanetary transfers [14]–[16] and quadcopters [12], [13] has shown that a network can directly learn the optimal state feedback from a large dataset of optimal trajectories. This has led to the term Guidance & Control Networks (G&CNETs). G&CNETs offer a direct mapping from states to raw control commands, they can be inferred at a low computational cost and they are very flexible since there is no need to recompute optimal trajectories.

In this paper, we improve past work on G&CNETs [13] to increase the quadcopter's flight speed, leading to four main contributions. We make the step from energy-optimal to time-optimal control and provide the loss values when training G&CNETs to approximate the corresponding optimal control policies. We introduce an adaptive scheme that can estimate the maximum angular velocity of the propellers and use it to adjust the commands to remain time-optimal. We develop a training method that allows for the network to output time-optimal raw control commands taking a horizon into account of the next two waypoints. We demonstrate that the two-waypoints network can deal with dynamic waypoint locations during the flight.

We structure the paper as follows. First, the quadcopter model, optimal control problem (OCP), imitation learning procedure and experimental setup are described (Sec.II). We then consider the task of learning and flying the time-optimal OCP and show how control policies become more difficult to learn as we shift from the energy- to time-optimal control problem (Sec.III). As one flies more time-optimally, the rotors of the quadcopter saturate more, i.e. the control policy approaches 'bang-bang' control. Given this, we assess the importance of training G&CNETs at a maximum RPM limit that the rotors of the quadcopter can reach during flight. We also propose an algorithm that can identify the current maximum RPM limit in combination with an adaptive G&CNET which changes its control policy based on the identified limit (IV). Finally, we improve fast quadcopter flight by extending previous research on G&CNETs from single waypoint to consecutive waypoints flight (Sec.V). The resulting controller is also compared to another benchmark using time-optimal minimum snap trajectories [11].

Fig.1 shows one lap on a $4 \times 3\mathrm{m}$ track using a G&CNET which is trained to take two consecutive waypoints into account.

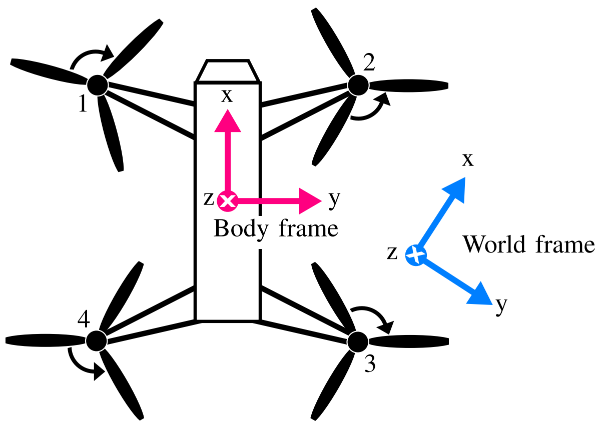


Fig. 2. Coordinate frames (Body x-axis points to the front of the drone).

II. Methodology

A. Quadcopter model

In this work, we use two coordinate frames as defined in Fig.2 and a quadcopter model with 19 states and 4 control inputs:

$$\mathbf{x} = [\mathbf{p}, \mathbf{v}, \lambda, \Omega, \omega, \mathbf{M}_{ext}]^T \quad \mathbf{u} = [u_1, u_2, u_3, u_4]^T$$

The state vector $\mathbf{x}$ contains the position $\mathbf{p} = [x, y, z]$ and velocity $\mathbf{v} = [v_x, v_y, v_z]$ which are both defined in the world frame. The Euler angles $\lambda = [\phi, \theta, \psi]$ which specify the orientation of the body frame, the angular velocities $\mathbf{\Omega} = [p, q, r]$ in the body frame, the propeller rates $\omega = [\omega_1, \omega_2, \omega_3, \omega_4]$ and external moments disturbances $\mathbf{M}_{ext} = [M_{ext,x}, M_{ext,y}, M_{ext,z}]$ [13]. The control inputs $\mathbf{u} = [u_1, u_2, u_3, u_4]$ are bounded $u_i \in [0, 1]$, such that $u_i = 0$ and $u_i = 1$ correspond to the minimal (ω_{min}) and maximal rotational speed (ω_{max}) of the corresponding propeller, respectively. Specifying the equations of motion (Eq.1) as:

$$f(\mathbf{x}, \mathbf{u}) = \begin{cases} \dot{\mathbf{p}} = \mathbf{v} \\ \dot{\mathbf{v}} = \mathbf{g} + R(\lambda)\mathbf{F} \\ \dot{\lambda} = Q(\lambda)\mathbf{\Omega} \\ I\dot{\mathbf{\Omega}} = -\mathbf{\Omega} \times I\mathbf{\Omega} + \mathbf{M} + \mathbf{M}_{ext} \\ \dot{\omega} = ((\omega_{max} - \omega_{min})\mathbf{u} + \omega_{min} - \omega)/\tau \\ \dot{\mathbf{M}}_{ext} = 0 \end{cases} \tag{1}$$

where $I = \mathrm{diag}(I_x, I_y, I_z)$ is the moment of inertia matrix and $\mathbf{g} = [0, 0, g]^T$ with $g = 9.81\ \mathrm{m\,s^{-2}}$ is the acceleration due to gravity. The rotational matrix $R(\lambda)$ transforms from the body to the world frame. We use the notation c_θ and s_ϕ to denote the cosine and sine of the corresponding Euler angle, respectively.

$$R(\lambda) = \begin{bmatrix} c_\theta c_\psi & -c_\phi s_\psi + s_\phi s_\theta c_\psi & s_\phi s_\psi + c_\phi s_\theta c_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & -s_\phi c_\psi + c_\phi s_\theta s_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix}$$

and $Q(\lambda)$ is the inverse transformation matrix:

$$Q(\lambda) = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta \end{bmatrix}$$

The forces $\mathbf{F} = [F_x, F_y, F_z]^T$ are computed using the thrust and drag model from [17]. Note that the superscript $\square^B$ denotes the body frame, all model parameters are listed in Tab.I.

$$\begin{aligned} F_x &= -k_x v_x^B \sum_{i=1}^{4} \omega_i \quad F_y = -k_y v_y^B \sum_{i=1}^{4} \omega_i \\ F_z &= -k_\omega \sum_{i=1}^{4} \omega_i^2 - k_z v_z^B \sum_{i=1}^{4} \omega_i - k_h (v_x^{B2} + v_y^{B2}) \end{aligned} \tag{2}$$

and the moments $\mathbf{M} = [M_x, M_y, M_z]^T$ are defined as:

$$\begin{aligned} M_x &= k_p(\omega_1^2 - \omega_2^2 - \omega_3^2 + \omega_4^2) + k_{pv} v_y^B \\ M_y &= k_q(\omega_1^2 + \omega_2^2 - \omega_3^2 - \omega_4^2) + k_{qv} v_x^B \\ M_z &= k_{r1}(-\omega_1 + \omega_2 - \omega_3 + \omega_4) \\ &\quad + k_{r2}(-\dot{\omega}_1 + \dot{\omega}_2 - \dot{\omega}_3 + \dot{\omega}_4) - k_{rr} r \end{aligned} \tag{3}$$

We utilize an adaptive method proposed in [13], which accounts for model mismatches in the moment equations (Eq.3) to make the G&CNET more robust. The idea is to use domain randomization during the learning process by assuming constant external moment disturbances for each optimal trajectory. The difference between the measured and modelled moments can then be computed onboard and fed to the G&CNET.

B. The optimal control problem

The cost function $J(\mathbf{u}, T)$ (Eq.4) minimizes two objectives: the total time of flight T and the energy $\int_0^T ||\mathbf{u}(t)||^2 dt$. Both objectives are weighed using the hybridisation parameter ϵ, such that $\epsilon = 1$ corresponds to energy-optimal flight and $\epsilon = 0$ corresponds to time-optimal flight. While not directly minimizing for time, the energy-optimal term is useful as it creates smooth control inputs which leave more room for errors compared to the fully time-optimal 'bang-bang' control profile. Denoting X as the state space and U as the set of admissible controls. The optimal control problem considered tries to find the optimal control policy $\mathbf{u} : [0,1] \to U$ such that the quadcopter is steered from initial conditions $\mathbf{x}_0$ to a set of final conditions S, while minimizing the cost function $J(\mathbf{u}, T)$:

$$\begin{aligned} \underset{\mathbf{u},T}{\text{minimize}} \quad & J(\mathbf{u},T) = (1-\epsilon)T + \epsilon \int_0^T ||\mathbf{u}(t)||^2 dt \\ \text{subject to} \quad & \dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) \\ & \mathbf{x}(0) = \mathbf{x}_0 \\ & \mathbf{x}(T) \in S \end{aligned} \tag{4}$$

We follow a similar procedure as in [12] by transcribing the OCP into a Nonlinear Programming (NLP) problem using the Hermite Simpson collocation method. The NLP problem is formulated with the modelling language AMPL [19], discretized into $N+1$ points and solved using a sequential quadratic programming solver called SNOPT [20]. AMPL is used as it allows to inform SNOPT on the gradients and Hessian of the problem, making it easier for the solver to converge. We denote the resulting optimal trajectory as $\mathbf{x}_0^* \dots \mathbf{x}_N^*$ with corresponding optimal controls $\mathbf{u}_0^* \dots \mathbf{u}_N^*$.

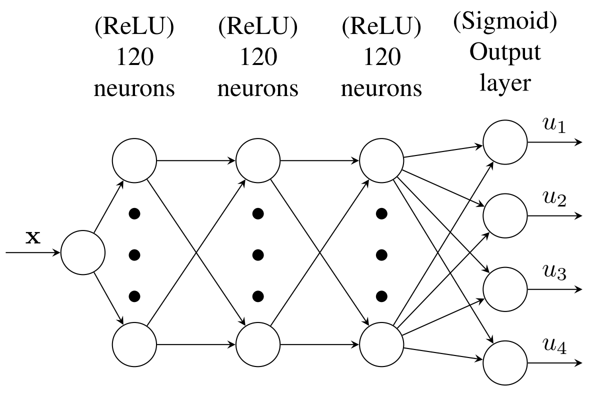


Fig. 3. Feed forward network architecture.

C. The dataset and learning procedure

The G&CNETs in this work are trained via imitation / supervised learning using datasets of optimal state-action pairs, where the states serve as features and the controls as labels. Hence each entry in the dataset is $(\mathbf{x}_i^*, \mathbf{u}_i^*) \quad i = 0, \dots, N$. Learning the optimal state feedback policy G&CNET$(\mathbf{x}_i^*) \approx \mathbf{u}_i^*$ is possible because of the existence and uniqueness of an optimal state-feedback which is a result of the Hamilton-Jacobi Bellman equations [14]. The general network architecture used throughout this work is a feed-forward neural net as depicted in Fig.3. We use the mean squared error as loss function:

$$\mathcal{L} = ||\text{G\&CNET}(\mathbf{x}_i^*) - \mathbf{u}_i^*||^2$$

The dataset is split into training data (80%) and validation data (20%). The weights of the network are updated using the Adam optimizer [21] without weight decay. The starting learning rate is set to $l = 0.1 \cdot 10^{-2}$ and a scheduler is used to reduce the learning rate by a factor of $f = 0.9$ whenever the loss on the validation dataset plateaus for $p = 6$ epochs. In order to facilitate the learning process the features are normalized and an addition output layer is added at the end to map the control inputs $u_i \in [0,1]$ to the corresponding RPMs.

D. Experimental setup

The experimental platform in this work is the Parrot Bebop 1 quadcopter in combination with the open-source Paparazzi UAC software [22]. This drone has been designed for the selfie-drone market, hence it is not a racing drone. Having a maximum thrust-to-weight ratio of ~ 1.7 it is not able to reach the same speeds as drones in the autonomous drone racing literature. However, this also means that it will allow

k_x	k_y	k_ω	k_z	k_h	I_x	I_y	I_z
1.08e-05	9.65e-06	4.36e-08	2.79e-05	6.26e-02	0.000906	0.001242	0.002054

k_p	k_{pv}	k_q	k_{qv}	k_{r1}	k_{r2}	k_{rr}	τ
1.41e-09	-7.97e-03	1.22e-09	1.29e-02	2.57e-06	4.11e-07	8.13e-04	0.03

TABLE I
MODEL PARAMETERS FOR THE PARROT BEBOP 1 QUADCOPTER. THE MOMENTS OF INERTIA HAVE BEEN TAKEN FROM [18], ALL OTHER PARAMETERS FROM [13].

us to reach saturation of its controls in the relatively small flight space at our disposal and that it is safer to test with. These latter properties were the main motivation for selecting the Parrot Bebop 1 for this study. The Bebop is equipped with an MPU6050 IMU and a Parrot P7 dual-core Cortex A9 CPU which we use to run our code in real-time onboard the drone. In addition, the quadcopter can measure the angular velocities of the four propellers. We fly the quadcopter at the faculty of Aerospace Engineering (TU Delft) inside The Cyberzoo, which is a 10-by-10 meter laboratory equipped with a motion-capture system (Optitrack). We send position, velocity and attitude measurements to the drone in real-time and fuse these with the IMU measurements using an Extended Kalman filter to provide accurate state estimation to the G&CNET. The G&CNET then directly outputs RPM commands which are sent to the motors. In the case of the differential-flatness-based-controller we use an INDI controller to track the reference trajectory.

III. TIME-OPTIMAL QUADCOPTER FLIGHT

In this section, we train G&CNETs to approximate different control policies, from fully energy-optimal ($\epsilon = 1.0$) to fully time-optimal ($\epsilon = 0.0$). We simulate the response of the G&CNET with $\dot{\mathbf{x}} = f(\mathbf{x}, \text{G\&CNET}(\mathbf{x}))$ and show that for close to time-optimal flight, reaching a low loss $\mathcal{L}$ not only becomes more difficult but it is also more critical as the quadcopter has less control authority to recover from errors compared to energy-optimal flight.

The datasets and network architecture

We generate all training datasets by uniformly sampling the initial conditions for each trajectory in the following bounds (where the target waypoint is the reference, see the top figure of Fig.19 in App.VII-C as an example):

$$
\begin{aligned}
x_0 &\in [-5.0, -2.0], \mathrm{m} & y_0 &\in [-1.0, 1.0], \mathrm{m} \\
z_0 &\in [-0.5, 0.5], \mathrm{m} & v_{x_0} &\in [-0.5, 5.0], \mathrm{m\,s^{-1}} \\
v_{y_0} &\in [-3.0, 3.0], \mathrm{m\,s^{-1}} & v_{z_0} &\in [-1.0, 1.0], \mathrm{m\,s^{-1}} \\
\phi_0 &\in [-40, 40], \mathrm{deg} & \theta_0 &\in [-40, 40], \mathrm{deg} \\
\psi_0 &\in [-60, 60], \mathrm{deg} & p_0 &\in [-1, 1], \mathrm{rad\,s^{-1}} \\
q_0 &\in [-1, 1], \mathrm{rad\,s^{-1}} & r_0 &\in [-1, 1], \mathrm{rad\,s^{-1}}
\end{aligned}
$$

The RPM limits are set to $\omega_{min} = 3000$ and $\omega_{max} = 12000$, respectively, and the initial rotational speeds are sampled in:

$$\omega_{i_0} \in [-\omega_{min}, \omega_{max}], \text{RPM} \quad i = 1, \dots, 4$$

For each trajectory we assume constant external moment disturbances sampled in these bounds:

$$
\begin{aligned}
M_{ext,x} &\in [-0.04, 0.04], \mathrm{N\,m} \\
M_{ext,y} &\in [-0.04, 0.04], \mathrm{N\,m} \\
M_{ext,z} &\in [-0.01, 0.01], \mathrm{N\,m}
\end{aligned}
$$

We set the desired final states for each trajectory to: $\mathbf{p}_f = \mathbf{0}$ m, $\psi_f = 45°$, $\mathbf{\Omega}_f = \mathbf{0}$ $\mathrm{rad\,s^{-1}}$ and $\dot{\mathbf{\Omega}}_f = \mathbf{0}$ $\mathrm{rad\,s^{-2}}$. The final velocity $\mathbf{v}_f$ is constrained such that its direction coincides with the final heading $\psi_f = 45°$, the final velocity magnitude and all remaining states are left free. These specific constraints are chosen as they allow the G&CNET to fly a variety of tracks containing only turns to the right by moving the target waypoint right before the quadcopter reaches it. Sec.V goes into more detail regarding single and consecutive waypoints flight. Each dataset contains 10,000 optimal trajectories which are all sampled in $N = 199$ points. We choose to train each individual G&CNET for $p = 10$ epochs with a training batch size of 256. Fig. 4 shows how a typical optimal control input u^* for one of the four rotors varies for different ϵ. Given the same initial and final conditions, u^* varies from a smooth control profile ($\epsilon = 1.0$) to a so-called 'bang-bang' control profile ($\epsilon = 0.0$) for fully time-optimal flight, where at all times at least one of the four rotors is saturating. Note that the time-optimal solution completes this trajectory in 1.2s compared to 1.65s for the energy-optimal case.

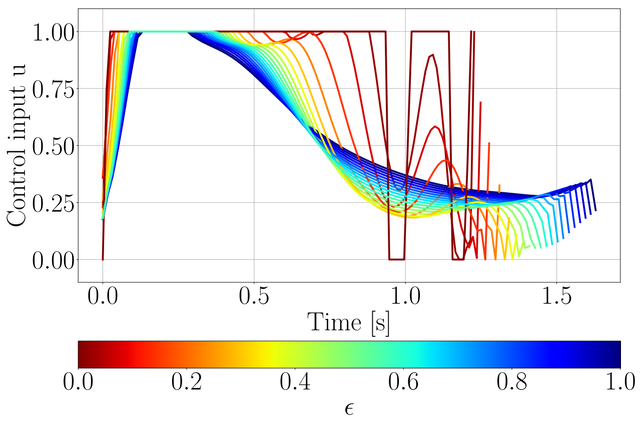


Fig. 4. Optimal control input u^* for different ϵ (only one rotor is shown). All trajectories have the same initial and final conditions.

Results & Discussion

We report the resulting loss $\mathcal{L}$ and corresponding mean control error $[\%]$ in Tab. II. Clearly, as we weigh the time-optimal objective in Eq. 4 more heavily by decreasing ϵ, the loss $\mathcal{L}$ goes up. The reason the time-optimal policy is more difficult to learn is the high number of switching times as depicted in Fig. 4. The large gradients of the resulting topology are more difficult to approximate precisely than the smooth continuous control profile for energy-optimal flight.

In practice, this issue can be mitigated by increasing the training dataset size, training for longer and eventually increasing the size of the network. Though one must mention that a larger network will directly impact the frequency at which the network can be inferred onboard the drone. In addition, for time-optimal flight, bringing the loss $\mathcal{L}$ down will not solve all problems. Time-optimal flight means the drone operates at the edge of its flight envelope, leaving no room for control authority. This means that recovering from small control errors becomes much more difficult than for energy-optimal flight.

We test all G&CNETs in simulation by numerically integrating $\dot{\mathbf{x}} = f(\mathbf{x}, \text{G\&CNET}(\mathbf{x}))$ using Scipy (explicit Runge-Kutta integration method of order 5 [23]). No perturbations are added to these simulations and the same equations of motion and actuator delay as the ones used to solve the optimal trajectories are used. The solver chooses the step size and no zero-order hold is implemented. Even with these conditions, it becomes hard to maintain stable flight for $\epsilon = 0.1$ and $\epsilon = 0.0$. This suggests that for close to time-optimal flight, a training loss of $\mathcal{L} = 1.12 \cdot 10^{-3}$ (mean control error of $\pm 3.35\%$) is an upper bound to maintain stability.

ϵ	Loss $\mathcal{L}$	Control error [%]
1.0	$1.24 \cdot 10^{-4}$	± 1.12
0.95	$1.41 \cdot 10^{-4}$	± 1.19
0.9	$1.38 \cdot 10^{-4}$	± 1.18
0.85	$1.35 \cdot 10^{-4}$	± 1.16
0.8	$1.46 \cdot 10^{-4}$	± 1.21
0.75	$1.48 \cdot 10^{-4}$	± 1.22
0.7	$1.60 \cdot 10^{-4}$	± 1.27
0.65	$1.72 \cdot 10^{-4}$	± 1.31
0.6	$1.97 \cdot 10^{-4}$	± 1.40
0.55	$1.79 \cdot 10^{-4}$	± 1.34
0.5	$2.05 \cdot 10^{-4}$	± 1.43
0.45	$2.37 \cdot 10^{-4}$	± 1.54
0.4	$2.99 \cdot 10^{-4}$	± 1.73
0.35	$2.96 \cdot 10^{-4}$	± 1.72
0.3	$3.39 \cdot 10^{-4}$	± 1.84
0.25	$4.82 \cdot 10^{-4}$	± 2.20
0.2	$5.25 \cdot 10^{-4}$	± 2.29
0.15	$7.08 \cdot 10^{-4}$	± 2.66
0.1	$1.12 \cdot 10^{-3}$	± 3.35
0	$7.01 \cdot 10^{-3}$	± 8.37

TABLE II
MEAN SQUARED ERRORS ON VALIDATION DATASETS AND CORRESPONDING CONTROL ERRORS FOR G&CNETS TRAINED ON DIFFERENT COST FUNCTIONS.

IV. ACCOUNTING FOR THE VARYING MAXIMUM ANGULAR VELOCITY OF PROPELLERS

In this section, we investigate the importance of flying at a reachable maximum RPM limit ω_{max} by looking at how under- and overshooting this limit affects the robustness of the flight. The maximum angular velocity of rotors can either be wrongly identified in the first place, or momentarily change during flight due to varying aerodynamic load on the propellers or even decrease over time as the battery drains out. We have observed a steady drop of $\dot{\omega}_{max} = -1$ RPMs^{-1} on the Parrot Bebop 1 over a test flight of 6min (See Appendix VII-B). Time-optimal flight is characterised by control profiles that saturate the rotors for a considerable portion of time, consider Fig. 5 which shows the commanded and observed angular velocities of one rotor during a real flight with $\epsilon = 0.35$. The upper RPM limit $\omega_{max} = 12000$ cannot be reached. This begs the question, how crucial is it to fly at the correct ω_{max}?

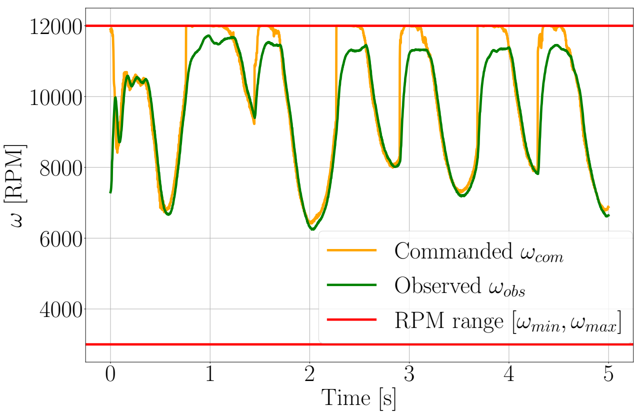


Fig. 5. Discrepancy between commanded and observed angular velocity ω during a real flight with $\epsilon = 0.35$ (only one rotor is shown).

Let's consider a range of $\omega_{max}[10000, 12000]$ RPM. Fig. 6 shows how the optimal control solution differs for different ω_{max} in the case of a time-optimal ($\epsilon = 0.0$) landing. The quadcopter starts from hover at a height of 5m and needs to reach the following final conditions: $\mathbf{p}_f = \mathbf{0}$, $\mathbf{v}_f = \mathbf{0}$, attitude $\lambda_f = \mathbf{0}$ and angular rates $\mathbf{\Omega}_f = \mathbf{0}$. No external moment disturbances $\mathbf{M}_{ext}$ are applied in this case. Given the symmetry of this OCP, the optimal control solution for all four rotors is the same. In the case where $\omega_{max} = 12000$ the rotors start saturating 0.1s later than in the case where $\omega_{max} = 10000$. Since the optimal control solutions here consist in applying the maximal breaking force from a certain switching point onwards until the end, overestimating ω_{max} will always result in a crash since the quadcopter will start breaking too late. This case also suggests that having the observed angular velocities of the propellers ω as state feedback is not sufficient to mitigate the effects of incorrectly identifying ω_{max} because when the drone observes that the rotors do not reach the desired ω_{max}, it is already too late. Finally, precise switching times between minimal and maximal control inputs are required to maintain stable flight, especially because little to no control authority is left as ϵ approaches zero.

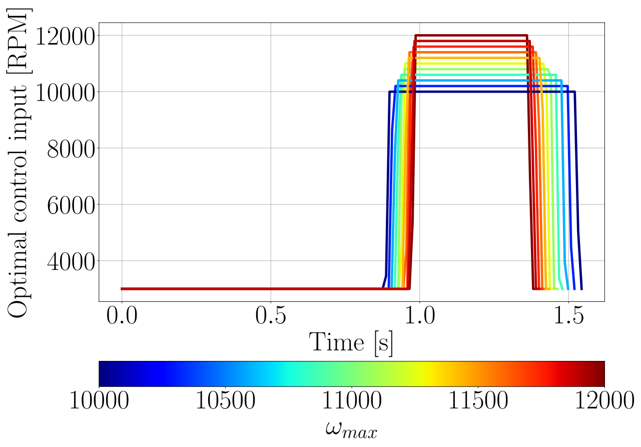


Fig. 6. Optimal control input [RPM] for different ω_{max} (only one rotor is shown). All trajectories have the same initial and final conditions. The goal is to perform a time-optimal ($\epsilon = 0.0$) landing from a height of 5m. Given the symmetry of this OCP all rotors receive the same control input.

Peak tracker algorithm

In the case where the initial guess for ω_{max} is set too high, the new limit can be identified onboard as soon as it is observable. We propose a peak tracker algorithm that sets ω_{max} to the highest observed ω_{obs} whenever the integral $\int_{t-\Delta t}^{t}(\omega_{exp} - \omega_{obs})dt$ surpasses a certain threshold p_{thresh}, see $t = 0.21$s in Fig. 7. The expected ω_{exp} can be computed onboard the drone by taking the first order delay of the commanded ω_{com} (Eq.1). Depending on the RPM range $[\omega_{min}, \omega_{max}]$ the parameters of this algorithm to be tuned are: the time window Δt over which the integral is computed and the last peak in observed ω_{obs} is recorded and the threshold p_{thresh} which triggers a change in ω_{max}. One should note that another obvious approach would be to model ω_{max} as a function of the quadcopter states $\mathbf{x}$ and its battery voltage. However, this would require system identification every time one changes the drone. The advantage of our algorithm is that it can easily be used for any drone, so long as the angular velocity of the propellers can be measured onboard. In addition, our algorithm could account for lower ω_{max} due to any unexpected failure that cannot be modelled. The current setup can only correct ω_{max} after overshooting it, otherwise, the limit is not observable. We tested the RPM peak tracker algorithm both in simulation and on the real quadcopter. In both cases, when the initial ω_{max} is too high, it takes less than $\Delta t = 0.13$s after one of the rotors saturates to correct the limit. We also simulated whether the peak tracker can continuously correct for a slowly decreasing ω_{max} (due to for instance the decrease in battery voltage). The peak tracker can keep the error between the identified limit and the correct ω_{max} below 70 RPM at all times.

Adaptive G&CNET

We considered training a G&CNET on a range of optimal trajectories with different ω_{max}. Instead of only learning the mapping between states $\mathbf{x}$ and controls $\mathbf{u}$, we add ω_{max} as one of the features. This allows the G&CNET to adapt during flight by using the output of the peak tracker algorithm as an additional input, see the new architecture in Fig. 8.

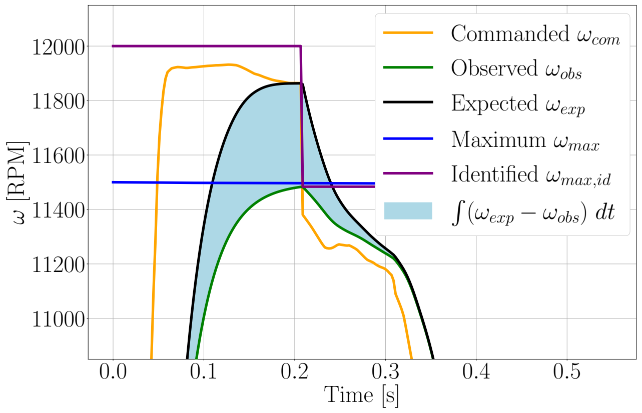


Fig. 7. Identifying the current ω_{max} (blue line) with the peak tracker algorithm.

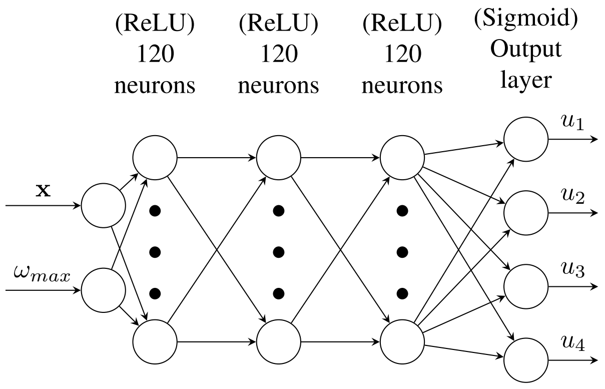


Fig. 8. Feed forward network architecture for an adaptive G&CNET to changes in ω_{max}.

Results and discussion

We first evaluate the effect of over- and undershooting ω_{max} in simulation. We train two adaptive G&CNETs using the same bounds for the initial and final conditions as in Sec.III except that each training dataset now contains 100,000 optimal trajectories and $\epsilon = 0.4$. The first G&CNET is trained on optimal trajectories where ω_{max} is sampled uniformly in $[10000, 11000]$ RPM and the second G&CNET $[11000, 12000]$ RPM. Note that for this analysis, one could just as well train multiple G&CNETs, each specialized for only one value for ω_{max}. The adaptive G&CNETs make it easier to quickly see how changing ω_{max} affects its performance. We generate an evaluation dataset separately which contains 10,000 optimal trajectories, all of which use $\omega_{max} = 11100$ RPM. This ω_{max} value was chosen in order to simulate over a large range for over- and undershooting this limit while staying close to the real limit of the Parrot Bebop 1. Given that the optimal trajectories are not the true analytical optimal solutions but are solved with a direct method, one can expect considerable numerical noise due to integration errors between the nodes. To alleviate some of this noise, we augment all 10,000 optimal trajectories using a node-doubling technique. The OCPs considered here are too complex for SNOPT to converge for $N > 400$ nodes if no good initial guess is provided. Hence we solve the OCP for $N = 100$, then interpolate the solution $\mathbf{x}^*$ and $\mathbf{u}^*$ using quadratic splines and finally project the interpolant on a new grid of nodes (e.g. $N = 200$) to have a good initial guess for the solver. Repeating this process allowed us to generate a high-fidelity evaluation dataset of 10,000 trajectories, with $N = 1000$ nodes each (which translates to 5mm between two consecutive nodes for a 5m long optimal trajectory). Fig. 9 shows the mean position errors [cm] from these 10,000 optimal trajectories when simulating the response of the G&CNETs $\dot{\mathbf{x}} = f(\mathbf{x}, \text{G\&CNET}(\mathbf{x}))$ starting from the same initial conditions as the trajectories in the evaluation dataset. By manually changing the input ω_{max} to the network, we can simulate how this affects the deviation from the optimal trajectories. Even when the G&CNET knows the correct limit (boxplot in the center of Fig. 9) its mean position error is around 4cm, which is due to the nonzero loss during training. We see a similar trend for over- and undershooting ω_{max}, the larger the difference, the larger the mean position error. This suggests that incorrectly identifying ω_{max} impacts the robustness of the flight.

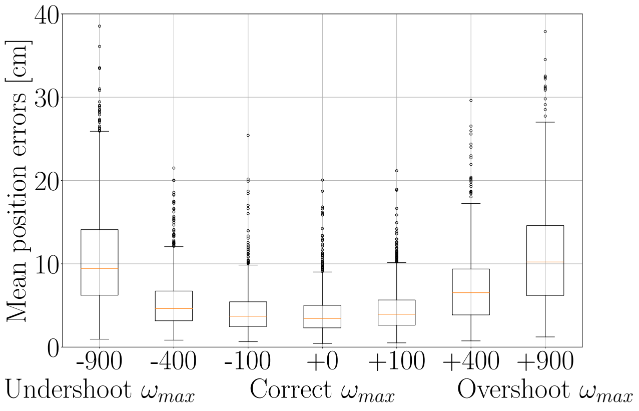


Fig. 9. Mean position errors [cm] from 10,000 optimal trajectories (1000 nodes per trajectory). For each boxplot the G&CNET either undershoots, overshoots or is exactly at the correct ω_{max} [RPM].

However, contrary to our expectations, even large deviations from ω_{max} are not the most critical contributor to the reality gap. In our experiments in simulation and on the real quadcopter, we were able to fly even when ω_{max} was off by $+1000$ RPM. Nevertheless, as we approach time-optimal flight, there is less room for error and flying as closely as possible to the optimal trajectory is relevant for robustness.

Consider Fig.10 which shows three real flights with the same G&CNET ($\epsilon = 0.5$). This G&CNET is trained on 60,000 optimal trajectories and ω_{max} is sampled uniformly in $[10500, 13000]$ RPM. We artificially limit the maximum angular velocity of the propellers to $\omega_{max} = 11000$ RPM. The trajectory on the left shows the resulting flight when the G&CNET receives the correct ω_{max} as input, the other two trajectories are flights where the G&CNET receives an incorrect ω_{max} as input (overshoot of $+500$ RPM and $+1000$ RPM, respectively). Overshoot refers to the G&CNET assuming that ω_{max} is higher than it actually is. The differences between these three cases are mostly visible during the first lap. The aggressive start of the G&CNET (from hover to pitch down of $\theta = -85°$) and the first turn deviate substantially from the optimal path the more one overshoots ω_{max}.

A way to fly more robustly is to use this same G&CNET in combination with the peak tracker algorithm. We set the initial guess for maximum RPM to $\omega_{max} = 12000$ and we do not artificially limit the maximum angular velocity of the propellers. In the case of the Bebop 1, the real physical limit of the propellers is $\omega_{max} = 11300$, hence we overshoot the limit by $+700$ RPM. It takes the peak tracker 0.2s after the start of the flight and 0.1s after the first rotor saturates to identify the correct limit ($\omega_{max} = 11300$) and feed it to the G&CNET. The resulting flight is shown in Fig.11. Despite initially overshooting the correct limit by $+700$ RPM the quadcopter does not deviate as much from the optimal path as is the case in Fig.10.

More research has to be done in this area, it is conceivable that for more aggressive flight (e.g. $\epsilon = 0.0$), overestimating ω_{max} becomes even more critical. Recall Fig.6, where saturating the rotors 0.1s too late would result in breaking 1m behind the optimal breaking point for a quadcopter travelling at $10\,\text{m}\,\text{s}^{-1}$. In addition, we learned that the range of values for ω_{max} one chooses to train a G&CNET on affects how well the network flies overall. For example, a G&CNET trained with ω_{max} uniformly sampled in $[10500, 13000]$ RPM will fly less consistent laps than a network with a smaller ω_{max} range (e.g. $[11000, 12000]$ RPM). This might indicate that the current network architecture needs to be revisited to learn the control policies more accurately. One can see in Fig.10,11 that the G&CNET struggles to fly through the first waypoint during the first laps.

V. Consecutive waypoints flight

The authors of [9] have analyzed the gaze of human drone pilots which showed that they looked at multiple gates in advance as opposed to only fixating on the next gate. Previous work [12], [13] focused on training and deploying G&CNET on quadcopters to fly from one point in space to a specific waypoint. By cleverly setting the final conditions of the OCP, it is possible to switch the position of the waypoint right before the quadcopter arrives to make it fly continuously. The obvious next step is to train the G&CNET on two consecutive waypoints. The global optimal trajectory would optimize for the full track, taking all waypoints into account at one. However, optimizing for a horizon of at least two waypoints will get the drone closer to optimality than a single waypoint approach. We propose one way in which two consecutive waypoints flight can be implemented and show some of the benefits of such a guidance strategy.

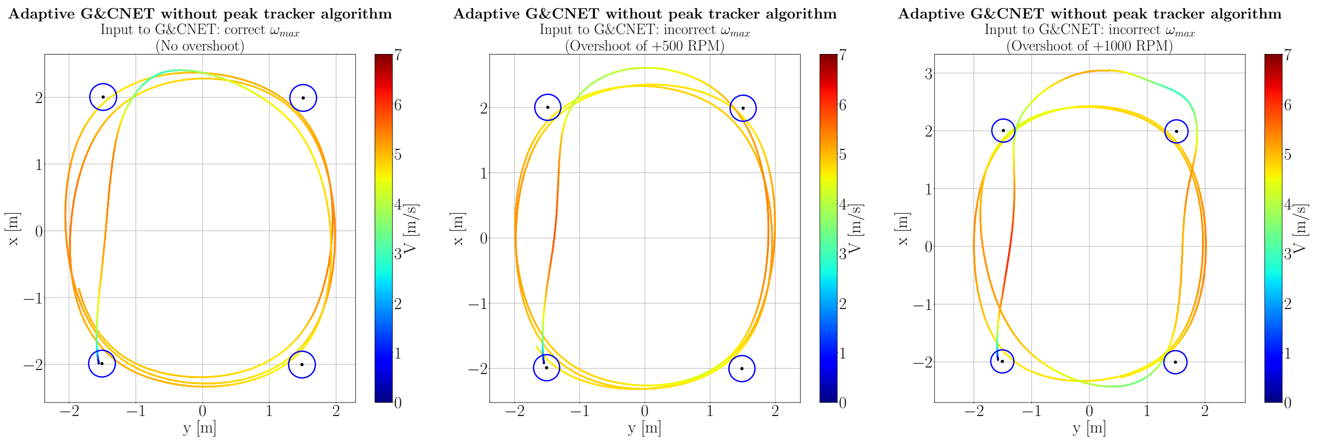


Fig. 10. Trajectories (top view) of a real flight with $\epsilon = 0.5$. Left: no ω_{max} overshoot. Center: ω_{max} overshoot of +500 RPM. Right: ω_{max} overshoot of +1000 RPM. During training, the optimal trajectories need to pass within a sphere of radius 20cm which is indicated by the blue circles, see Sec.V for details.

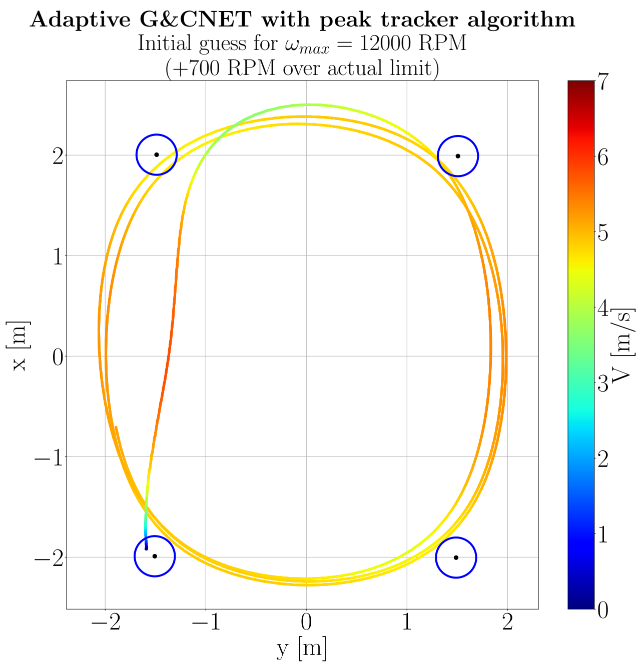


Fig. 11. Trajectory (top view) of a real flight with $\epsilon = 0.5$ using the peak tracker algorithm and an adaptive G&CNET. The initial guess for ω_{max} is off by +700 RPM. During training, the optimal trajectories need to pass within a sphere of radius 20cm which is indicated by the blue circles, see Sec.V for details.

Methodology to learn complex tracks

Consider the task of flying through a set of gates in a time-optimal fashion. The optimal approach to one gate depends heavily on the position and orientation of the next gate. To learn different optimal trajectories based on the relative position and orientation of two consecutive gates we add an intermediate constraint in AMPL. This constraint enforces the optimal solution to pass through a sphere that is centered at $\mathbf{WP}_{1,pos} = [\mathrm{WP}_{1,x}, \mathrm{WP}_{1,y}, \mathrm{WP}_{1,z}]$. Although we are not using the camera of the drone in this work, it is important that the drone's heading ψ always points in a direction such that the next waypoint is in the field of view of the camera. Hence we also add a term to enforce the heading $\mathrm{WP}_{1,psi}$ at the intermediate waypoint:

$$\mathrm{WP}_{1,threshold} \geq (x_i^* - \mathrm{WP}_{1,x})^2 + (y_i^* - \mathrm{WP}_{1,y})^2 + (z_i^* - \mathrm{WP}_{1,z})^2 + (\psi_i^* - \mathrm{WP}_{1,psi})^2$$

We choose this constraint because it allows to relax the position and heading error at the intermediate waypoint. It is difficult to know in advance what the optimal path and heading at any given stage of a complex track should be. Implementing the constraint this way gives the solver some freedom to pass through the intermediate waypoint in a more optimal way based on the initial conditions and the relative position of the two waypoints. The bottom figure in Fig.19 (App.VII-C) shows optimal trajectories with the intermediate waypoint constraint. Since the trajectories are now on average longer than for single waypoint flight we sample them in $N = 319$ points. We inform the G&CNET on the relative position of the two upcoming waypoints $\mathbf{WP}_{rel}$ by adding it as an input to the network architecture, see Fig.12. We do not add ω_{max} as additional input to the network architecture here as the size of the network is reaching its limit in terms of the amount of information it can carry. In order to always reach the desired saturation level, we fly conservatively at an ω_{max} that the quadcopter can reach.

This methodology lends itself well to learning to fly more complex tracks as different types of optimal turns can be generated. In Appendix VII-A we provide a trajectory of flight in simulation on a figure-eight track. The training dataset can either be tailored to a track if it is known in advance or as we will see at the end of this section, it is also possible to cover a range of relative positions between the two waypoints such that the G&CNET can fly different variations of tracks.

Comparison with energy-optimal single waypoint flight

Another advantage of learning to fly while taking more than one waypoint into account is that the G&CNET flies more optimally through the entire track. We show this by performing two real flights on the Bebop with energy-optimal

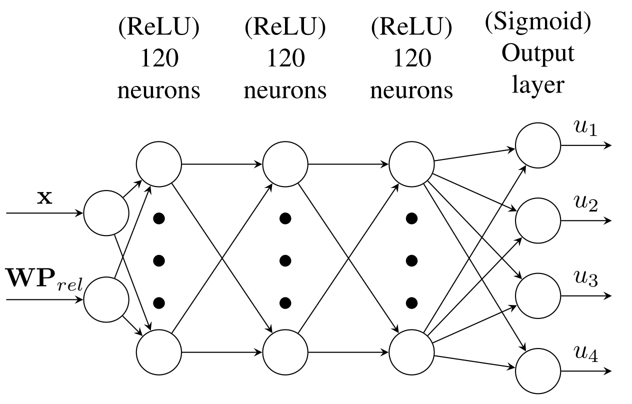


Fig. 12. Feed forward network architecture for G&CNETs using multiple waypoints. We add the input $\mathbf{WP}_{rel}$ which informs the network on the relative position of the two upcoming waypoints.

G&CNETs and subsequently computing the cost (Eq.4 with $\epsilon = 1.0$) over time. We choose the energy-optimal control problem here because it is the easiest optimal control policy for G&CNETs to learn. The same analysis could also have been done for a different ϵ. Fig.13 shows the resulting trajectories. The left figure shows the G&CNET which is only trained on a single waypoint. We switch to the next waypoint when the Euclidean distance in three dimensions between the quadcopter and the waypoint is below 1.2m [13]. We experimented with varying switching distances. In general, the larger the switching distance, the more the quadcopter will cut the corner. For smaller switching distances the position error from the waypoint becomes smaller, however, the quadcopter slows down as the G&CNET tries to perfectly meet the final conditions of the OCP. Note that we use a free final magnitude in velocity, only the direction of the velocity vector is constrained to be aligned with the desired final heading angle ($\psi_f = 45°$). The other G&CNET (center figure) is trained on two consecutive waypoints with WP$_{1,threshold} = 0.2$ and WP$_{1,psi} = 45°$. This means that optimal trajectories have to pass through a sphere of radius 20cm with a heading of roughly $\psi = 45°$. The G&CNET is trained on a dataset that contains two "types" of turns: trajectories where the two waypoints are 3m apart and trajectories where they are 4m apart. Hence only one extra input to the G&CNET is required ($\mathbf{WP}_{rel}$ in Fig.12) to inform the network on which of the two possible turns is upcoming. A visualization of optimal trajectories used to train G&CNETs in the case of single and consecutive waypoints is provided in App.VII-C. Compared to the single waypoint flight, we do not have to trade off position errors from the waypoint and speed anymore. We switch waypoints every time the G&CNET passes one, hence 3m before the final waypoint during the first turn and 4m before the final waypoint the next turn and so forth. We compute the cost $\int_0^T ||\mathbf{u}(t)||^2 dt$ over these flights and plot it in the right-most figure (Fig.13). The network that is trained on two consecutive waypoints spends less energy, hence minimizing the cost function better over time. We also note that the control inputs are smoother and saturate less in the case of consecutive waypoints flight, leaving more control authority to recover from errors. This is likely because the G&CNET trained on a single waypoint unnecessarily saturates the rotors because it decelerates and accelerates more.

Comparison with minimum snap benchmark

We also consider flying the $4\times3\text{m}$ track as fast as possible with the G&CNET. We choose the well-known differential-flatness-based-controller (DFBC) [11] as a benchmark to compare our G&CNET. This state-of-the-art controller uses polynomials to generate smooth trajectories by minimizing snap, the fourth derivative of position. The reference trajectory is then tracked by an outer-loop Incremental Nonlinear Dynamic Inversion (INDI) controller. Just as for the energy-optimal case, the G&CNET is trained on trajectories where the two waypoints are 3m apart and trajectories where they are 4m apart, only now $\epsilon = 0.5$. The resulting flights are shown in Fig.14. Since the G&CNET always tries to fly through the apex of the blue circles in Fig.14, we moved the waypoints in the polynomial generation for the DFBC inwards such that they coincide with these apexes to make the comparison fairer. Both controllers have their advantages, it is for instance possible to make more aggressive maneuvers with the G&CNET. This is especially noticeable in the first lap which the G&CNET performs in 3.22s (DFBC takes 3.46s). The DFBC however can sustain higher velocities once the transient behaviour at the start is over. The DFBC performs the second lap in 2.7s compared to 2.88s for the G&CNET. The commanded and observed angular velocities during these flights are provided in App.VII-D. The control inputs of the G&CNET are considerably smoother than the ones for the DFBC. One should note here that both controllers still have room for improvement. Weighted least squares could be implemented in the control allocation for the INDI used with the DFBC. In addition, the G&CNET has an unfair advantage in this comparison as it is trained for $\omega_{max} = 11300$ RPM which is close to the true limit of the Bebop. The DFBC however assumes that $\omega_{max} = 12000$ RPM. Finally, it is possible to fly faster with the G&CNET (lower ϵ) at the expense of larger position errors and more unstable flight.

Flexibility of G&CNETs

Finally, we highlight a major of advantage of G&CNET over the DFBC: the ability to recompute trajectories and the corresponding optimal controls online. G&CNETs are very flexible in so far as new optimal controls $\mathbf{u}^*$ are immediately computed even when deviating from the globally optimal path (so long as the state of the quadrotor has been represented closely enough in the training data set). This also means that the G&CNET can handle different waypoint positions within the training data. The DFBC on the other hand can only rely on the trajectory which has been generated offline. The DFBC will always try to stay as close as possible to this trajectory. This is a problem as deviations from this trajectory are bound to happen due to the reality

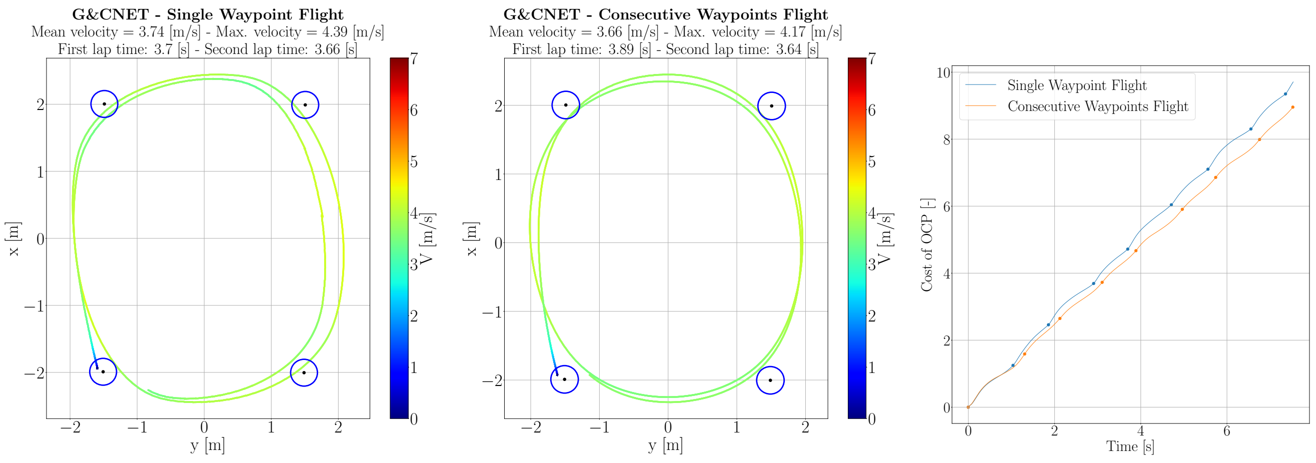


Fig. 13. Trajectories (top view) of a real flight with energy-optimal G&CNETs. Left: single waypoint flight. Center: consecutive waypoints flight. Right: cost function over time for both flights. The dots indicate the switching times to the next waypoint(s). The blue circles indicate the position constraint at the intermediate waypoint.

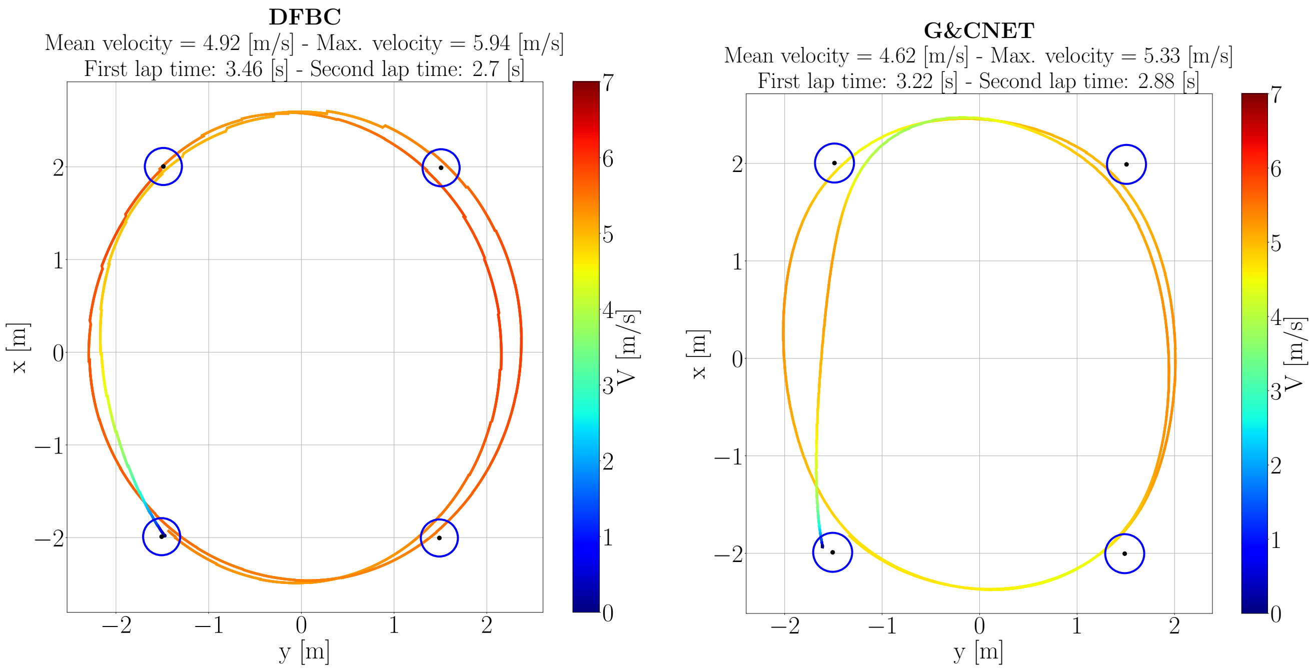


Fig. 14. Left: trajectory (top view) of a real flight with a DFBC (min snap). Right: trajectory (top view) of a real flight $\epsilon = 0.5$. The G&CNET is trained on 2 consecutive waypoints. The blue circles indicate the position constraint at the intermediate waypoint for the G&CNET.

gap. Once deviated this trajectory is no longer optimal and a new one should be computed. Moving a waypoint also requires computing a new optimal trajectory offline. Finally, due to its relatively small network size, the G&CNET can be inferred onboard the Bebop at a frequency of 450 Hz with the current network architecture. Both of these characteristics allow G&CNETs to cancel out approximations errors as opposed to accumulating these over time.

We consider the task of flying the 4×3m track, however, the four waypoints are now randomly moved (before the G&CNET takes them into account) within a square of 1m^2 in the XY plane (see dashed squares in Fig.15). We train a network on a range of relative waypoint positions by uniformly sampling $\mathrm{WP}_{1,x}, \mathrm{WP}_{1,y}$ in a square of 1m^2 centered at 3.5m from the final waypoint. The altitude is kept constant in this experiment for simplicity. Since the relative waypoint position can now vary in two dimensions, two extra inputs are required ($\mathbf{WP}_{rel}$ in Fig.12) to inform the network. We train two G&CNETs (one energy-optimal and one with $\epsilon = 0.5$) and fly these on the Bebop, the resulting trajectories are shown in Fig.15. In both cases, the networks manage to adapt their trajectory. The position errors from the waypoint become notably larger for the faster G&CNET as the policy is more difficult to learn (Sec.III) and the reality gap (hardware delays, state estimation errors and modelling errors) are harder to cope with when one flies more time-optimally.

A similar observation as in Sec.IV is made here regarding lower control accuracy when training the G&CNET on a larger range of data. We flew the $4 \times 3\text{m}$ track with fixed waypoints using the G&CNET that has learned to fly on a range of different waypoint positions, the G&CNET flies less consistent laps (Fig.16) compared to the G&CNET that is specifically trained on the $4 \times 3\text{m}$ track (Fig.14). Both networks roughly have the same network architecture (only one extra input neuron for the case where both $\text{WP}_{1,x}$ and $\text{WP}_{1,y}$ are varied). It is possible that the current network size needs to be increased for both G&CNETs to fly the $4 \times 3\text{m}$ track with the same accuracy.

VI. Conclusion

Guidance & Control Networks have been studied in the context of fast quadcopter flight. We showed that the control policies for the time-optimal control problem are considerably more difficult to learn than for the energy-optimal control problem. For close to time-optimal flight with the Bebop 1, average control errors of $\pm 3.35\%$ are already too high to maintain stable flight in simulation. We demonstrated that the maximum angular speed of propellers ω_{max} affects the switching times in the time-optimal control profile. We then went on to show that the more one over- or underestimates ω_{max}, the larger the mean the position error from the optimal trajectory becomes, which in turn affects the robustness of the flight. We propose a peak tracker algorithm to identify ω_{max} onboard in combination with a G&CNET that can adapt its control policy based on the identified value for ω_{max}. Our algorithm takes 0.1s after one of the four rotors saturates to identify the new limit, allowing it to stay close to the optimal trajectory in a real flight even when initially overestimating ω_{max} by $+700$ RPM. Finally, we extend previous work on G&CNETs by learning to fly while taking two upcoming waypoints into account. The new pipeline allows to generate training datasets that contain specific maneuvers for the G&CNET to learn, allowing it for instance to fly a figure-eight track in simulation. Compared to single-waypoint flight we optimize the energy-optimal cost function better over a $4 \times 3\text{m}$ track since the OCP formulation for multiple waypoints is more representative of the entire control task. We considered flying the $4 \times 3\text{m}$ track as fast as possible and benchmarking our G&CNET against the state-of-the-art differential-flatness-based-controller (DFBC). We show that G&CNETs can fly the track in similar lap times as the DFBC and adapt to varying waypoint positions. This highlights one of the main advantages of G&CNET compared to other optimality-based approaches, such as the DFBC: its flexibility to quickly recompute optimal control inputs.

Future work can be done on identifying the maximum angular velocity of each individual rotor, thereby not restricting propellers that are experiencing less aerodynamic load than the most limiting propeller. A more rigorous constraint could be implemented to make sure the waypoints are always in the field of view of the camera. The current dynamic model of the Bebop does not include the effects of downwash and errors in the thrust and drag model are common for flights where $\epsilon < 0.5$, hence one could consider using domain randomization in combination with onboard measurements to adapt to these model inaccuracies during flight. Finally, the methodology used in this work for two consecutive waypoints could be used to train a G&CNET on a much larger range of possible waypoint combinations, thereby allowing the network to fly a lot of different tracks.

Acknowledgment

The authors are grateful to Emmanuel Blazquez and Alexander Hadjiivanov for their valuable inputs and discussions throughout this project and to Erik van der Horst for his technical support. This research was co-funded under the Discovery programme of, and funded by, the European Space Agency.

References

[1] L. Bauersfeld and D. Scaramuzza, “Range, endurance, and optimal speed estimates for multicopters,” 2021. [Online]. Available: https://arxiv.org/abs/2109.04741

[2] C. De Wagter, F. Paredes-Vallés, N. Sheth, and G. de Croon, “Learning fast in autonomous drone racing,” *Nature Machine Intelligence*, vol. 3, no. 10, p. 923, 2021, copyright: Copyright 2021 Elsevier B.V., All rights reserved.

[3] G. Loianno, C. Brunner, G. McGrath, and V. Kumar, “Estimation, control, and planning for aggressive flight with a small quadrotor with a single camera and imu,” *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 404–411, 2017.

[4] A. Romero, S. Sun, P. Foehn, and D. Scaramuzza, “Model predictive contouring control for time-optimal quadrotor flight,” 2021. [Online]. Available: https://arxiv.org/abs/2108.13205

[5] K. Mohta, M. Watterson, Y. Mulgaonkar, S. Liu, C. Qu, A. Makineni, K. Saulnier, K. Sun, A. Zhu, J. Delmerico, K. Karydis, N. Atanasov, G. Loianno, D. Scaramuzza, K. Daniilidis, C. Taylor, and V. Kumar, “Fast, autonomous flight in gps-denied and cluttered environments,” *Journal of Field Robotics*, vol. 35, 12 2017.

[6] S. Li, M. M. Ozo, C. De Wagter, and G. C. de Croon, “Autonomous drone race: A computationally efficient vision-based navigation and control strategy,” *Robotics and Autonomous Systems*, vol. 133, p. 103621, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889020304619

[7] D. Mellinger, N. Michael, and V. Kumar, “Trajectory generation and control for precise aggressive maneuvers with quadrotors,” *The International Journal of Robotics Research*, vol. 31, no. 5, pp. 664–674, 2012. [Online]. Available: https://doi.org/10.1177/0278364911434236

[8] E. Kaufmann, A. Loquercio, R. Ranftl, M. Müller, V. Koltun, and D. Scaramuzza, “Deep drone acrobatics,” *RSS: Robotics, Science, and Systems*, 2020.

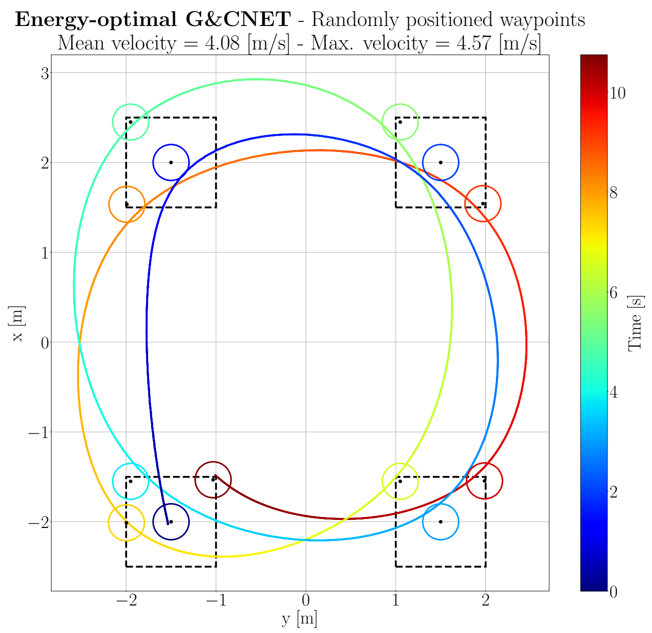
[9] P. Foehn, A. Romero, and D. Scaramuzza, “Time-optimal planning for quadrotor waypoint flight,” *Science Robotics*, vol. 6, no. 56, p. eabh1221, 2021. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abh1221

[10] T. Salzmann, E. Kaufmann, M. Pavone, D. Scaramuzza, and M. Ryll, “Neural-mpc: Deep learning model predictive control for quadrotors and agile robotic platforms,” 2022. [Online]. Available: https://arxiv.org/abs/2203.07747

[11] D. Mellinger and V. Kumar, “Minimum snap trajectory generation and control for quadrotors,” in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 2520–2525.

[12] S. Li, E. Öztürk, C. D. Wagter, G. C. H. E. de Croon, and D. Izzo, “Aggressive online control of a quadrotor via deep network representations of optimality principles,” *CoRR*, vol. abs/1912.07067, 2019. [Online]. Available: http://arxiv.org/abs/1912.07067

[13] R. Ferede, C. De Wagter, G. C. H. E. de Croon, and D. Izzo, “An adaptive control strategy for neural network based optimal quadcopter controllers,” Master’s thesis, TU Delft Aerospace Engineering, 2022. [Online]. Available: http://resolver.tudelft.nl/uuid:b43a9703-082c-47c7-a56e-d50794ee8c1c



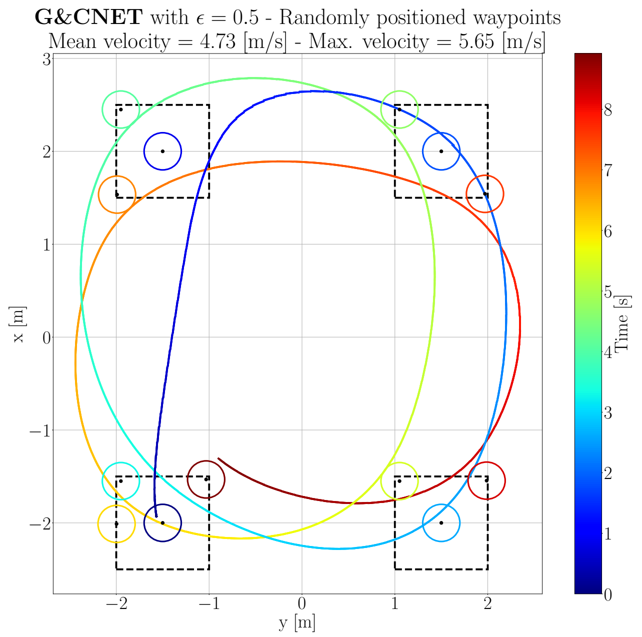


Fig. 15. Trajectories (top view) of a real flights with $\epsilon = 1.0$ (left) and $\epsilon = 0.5$ (right). The G&CNETs are trained on 2 consecutive waypoints with varying relative positions. We randomly position the waypoints within the dashed rectangles. The circles indicate the position constraint for each waypoint.

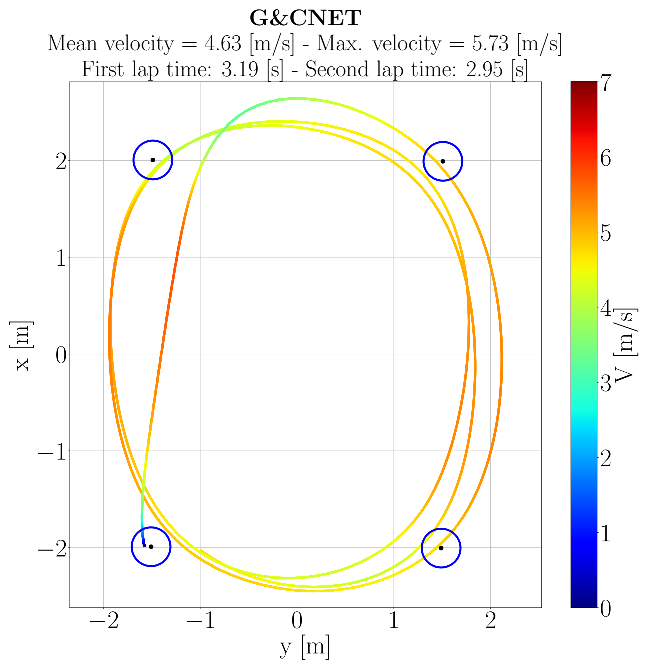


Fig. 16. Trajectory (top view) of a real flight $\epsilon = 0.5$. The G&CNET is trained on 2 consecutive waypoints with varying relative positions. The blue circles indicate the position constraint at the intermediate waypoint.

[14] C. Sánchez-Sánchez and D. Izzo, "Real-time optimal control via deep neural networks: Study on landing problems," *Journal of Guidance, Control, and Dynamics*, vol. 41, 10 2016.

[15] D. Izzo and E. Öztürk, "Real-time guidance for low-thrust transfers using deep neural networks," *Journal of Guidance, Control, and Dynamics*, vol. 44, no. 2, pp. 315–327, 2021.

[16] D. Izzo and S. Origer, "Neural representation of a time optimal, constant acceleration rendezvous," *Acta Astronautica*, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0094576522004581

[17] J. Svacha, K. Mohta, and V. R. Kumar, "Improving quadrotor trajectory tracking by compensating for aerodynamic effects," *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 860–866, 2017.

[18] S. Sun, C. C. de Visser, and Q. Chu, "Quadrotor gray-box model identification from high-speed flight data," *Journal of Aircraft*, vol. 56, no. 2, pp. 645–661, Mar. 2019. [Online]. Available: https://doi.org/10.2514/1.c035135

[19] R. Fourer, D. M. Gay, and B. W. Kernighan, "A modeling language for mathematical programming," *Management Science*, vol. 36, no. 5, pp. 519–554, 1990.

[20] P. E. Gill, W. Murray, and M. A. Saunders, "Snopt: An sqp algorithm for large-scale constrained optimization," *SIAM review*, vol. 47, no. 1, pp. 99–131, 2005.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[22] B. Gati, "Open source autopilot for academic research - the paparazzi system," in *2013 American Control Conference*, 2013, pp. 1478–1481.

[23] J. Dormand and P. Prince, "A family of embedded runge-kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 6, no. 1, pp. 19–26, 1980. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0771050X80900133

VII. Appendix

A. *Example of more complex track*

Fig.17 shows the trajectory of a flight in simulation using a G&CNET which learned to fly a figure-eight.

B. *Effect of battery on maximum RPM limit*

We tested a G&CNET with $\epsilon = 0.5$ on the Bebop by flying for 6min on a 4×3m track. Fig. 18 is a zoomed-in figure of rotor number 4 (see Fig. 2) which saturates most of the time since the flight path mostly consists of right turns. As the battery drains out ω_{max} decreases by roughly 1 RPMs^{-1}.

C. *Optimal trajectories for G&CNET training*

Fig.19 shows ten optimal trajectories for single and consecutive waypoints flight as can be found in the training datasets for G&CNETs.

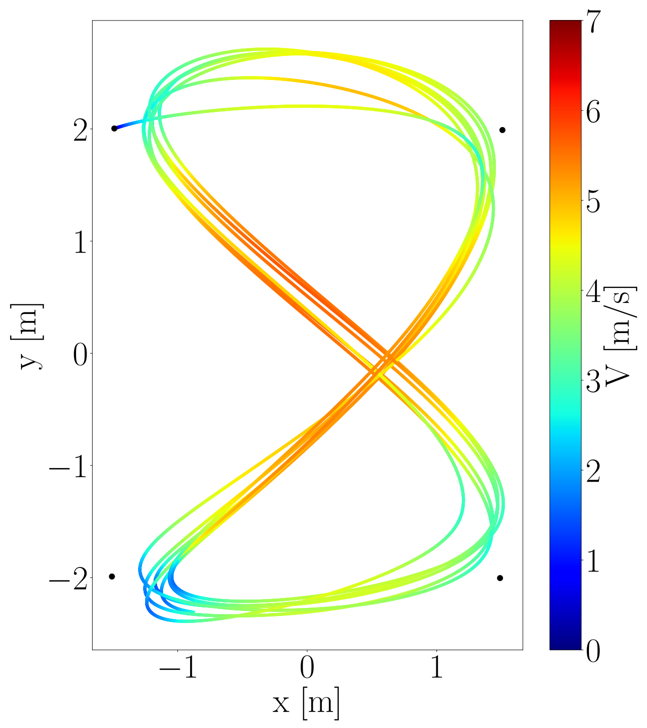


Fig. 17. Trajectory (top view) of a flight in simulation with $\epsilon = 0.4$. The G&CNET is trained on 4 sets of sharp turns. During the flight, we inform the G&CNET of the relative distance and angle of the two upcoming waypoints with two extra inputs.

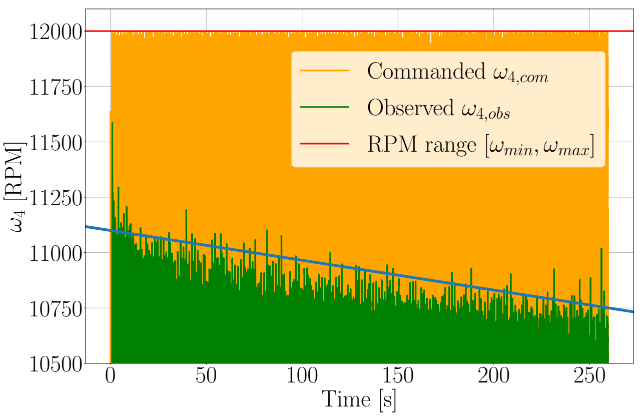


Fig. 18. Commanded and observed angular velocity ω during a test flight with $\epsilon = 0.5$ (only rotor number 4 is shown). The blue line roughly indicates the downward trend of ω_{max}.

D. Control inputs: DFBC vs G&CNET

Fig.20 shows the commanded and observed angular velocities of rotors for the real flights using a Differential-Flatness-Based-Controller (DFBC) and a Guidance & Control Network (see Sec.V).

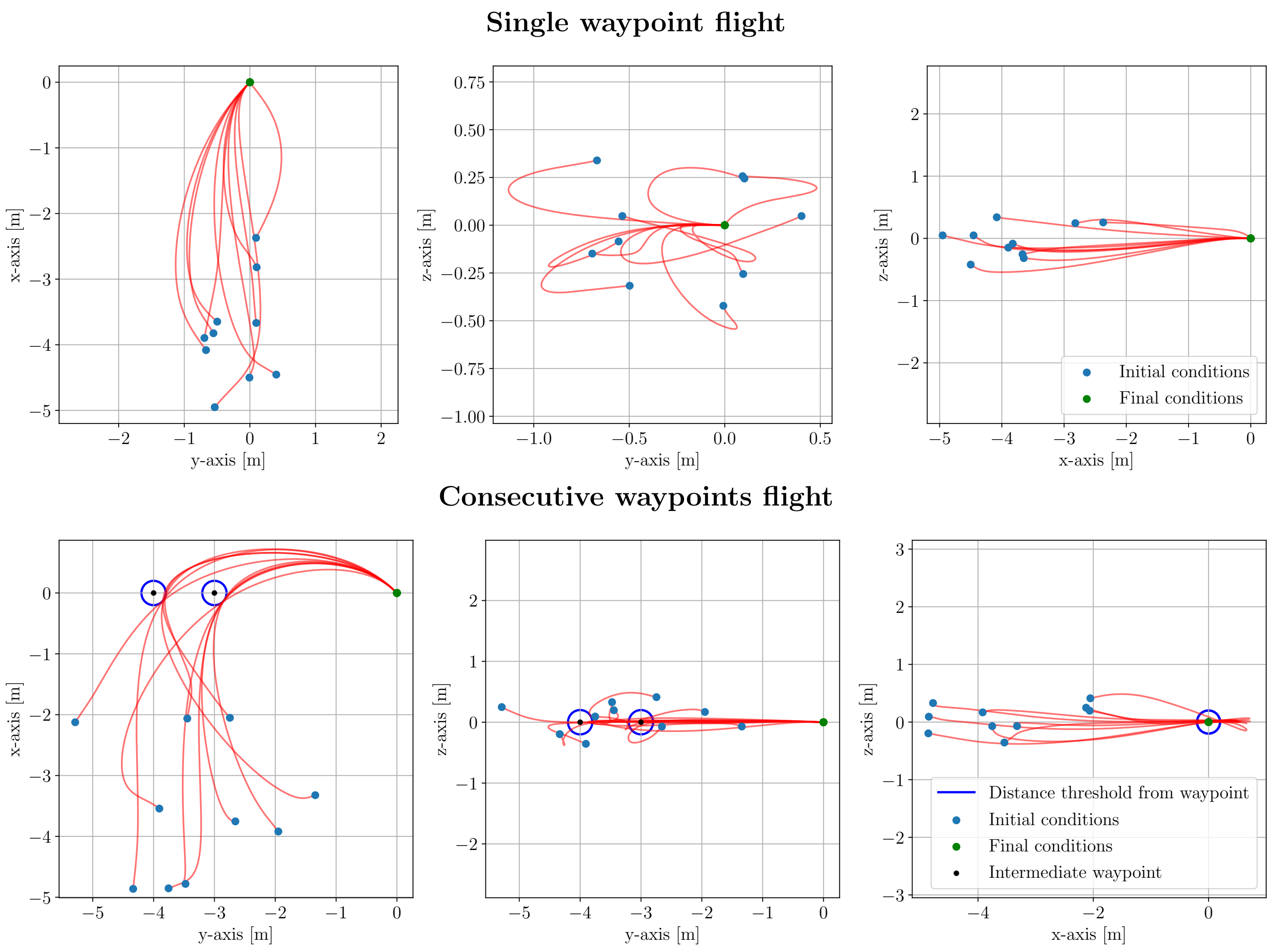


Fig. 19. Ten optimal trajectories for single and consecutive waypoints flight as can be found in the training datasets for G&CNETs.

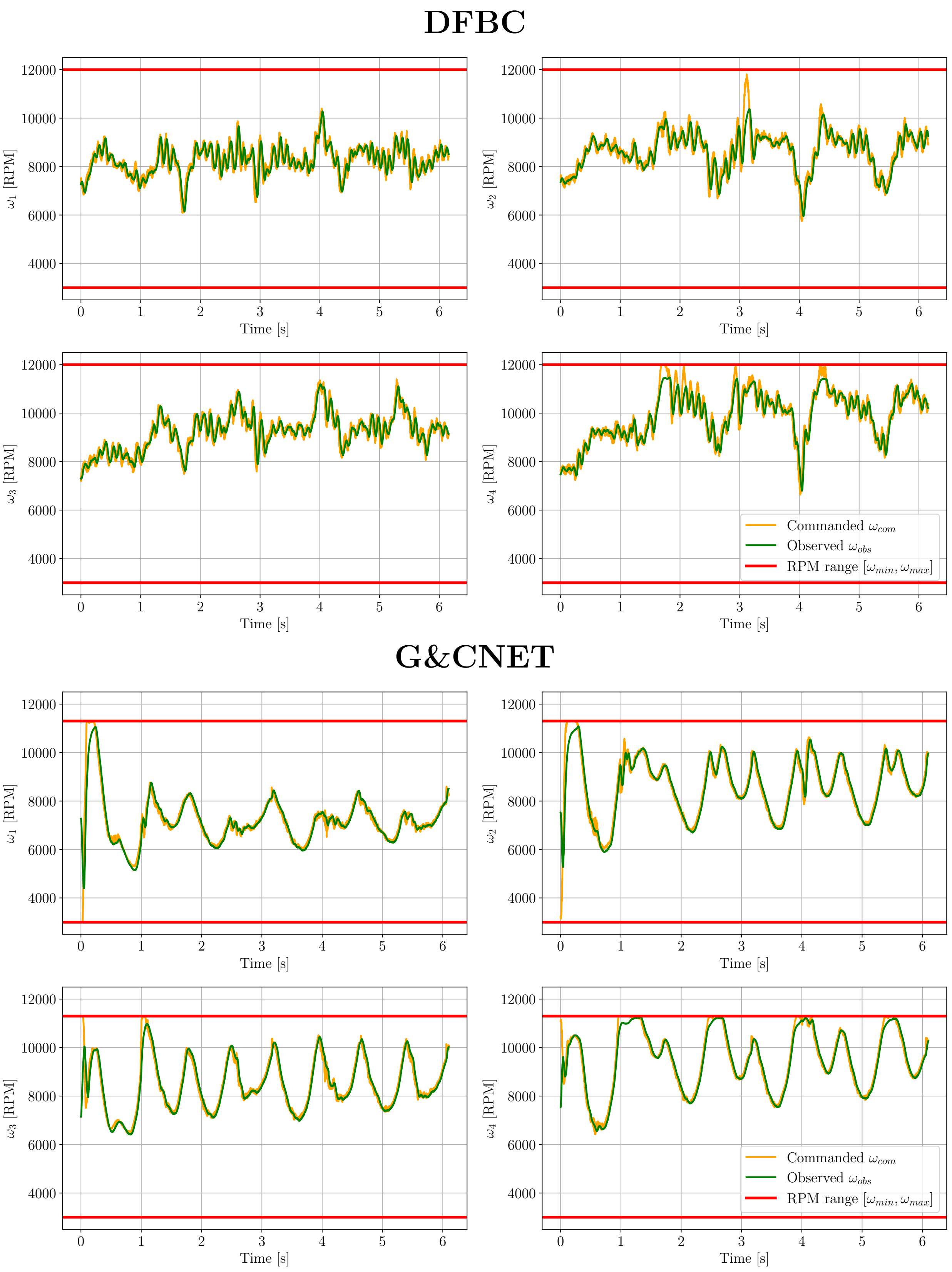


Fig. 20. Commanded and observed angular velocities of rotors for DFBC and G&CNET in SecV.

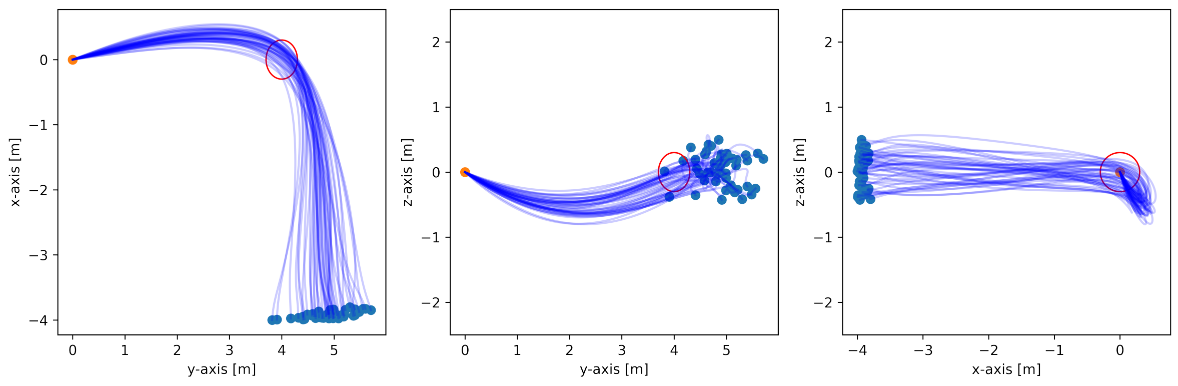
Part II

Literature review

Literature study

Delft University of Technology & European Space Technology and Research Centre

Neural Representation of Time-Optimal Quadrotor Flight



Student number	Surname, Given name
4662792	Origer, Sebastien

Thesis starting date:	February 7, 2022
Thesis Supervisors:	ir. C. De Wagter, Micro Air Vehicle Laboratory (TU Delft)
	Dr. D. Izzo, Advanced Concepts Team (ESA-ESTEC)
	Prof Dr. G. C. H. E. de Croon, Micro Air Vehicle Laboratory (TU Delft)

This document has been issued on February 27, 2023.

The title page figure shows a bundle of power-optimal trajectories passing through two consecutive waypoints.

TU Delft

Contents

I Introduction **2**

II Literature review **4**

1 Optimal Control Theory **5**
1.1 General optimal control problem formulation 5
1.2 History of optimal control theory 5
1.2.1 The Bellman equations 6
1.2.2 The Hamilton-Jacobi-Bellman equations 7
1.2.3 Pontryagin's Maximum Principle 8
1.3 Solving an optimal control problem 10
1.3.1 Indirect Methods 10
1.3.2 Direct Methods 11
1.3.3 Dynamic Programming 14
1.4 Closed loop control 14
1.4.1 Linear Quadratic Regulator (LQR) 14
1.4.2 Model Predictive Control (MPC) 14
1.5 Chapter Summary 16

2 Artificial Neural Networks **17**
2.1 Machine learning paradigms 17
2.1.1 Supervised learning 17
2.1.2 Unsupervised learning 17
2.1.3 Reinforcement learning 18
2.2 Types of artificial neural networks 18
2.2.1 Feedforward neural network 19
2.2.2 Recurrent neural network 20
2.3 Chapter Summary 20

3 Time-optimal Guidance & Control of Drones **21**
3.1 Trajectory tracking methods 21
3.2 Trajectory optimization methods 23
3.3 Chapter Summary 24

4 Bridging the Reality Gap for G&CNET **26**
4.1 Dynamics model & state estimation 26
4.2 Cost function 27
4.3 Other approach 28
4.4 Chapter Summary 28

III Thesis planning **29**

IV Research question **31**
4.5 Research Question 32
4.6 Research Objective 32

Bibliography **33**

I

Introduction

Introduction

Recent advances in computational power and commercial availability of micro aerial vehicles (MAV), such as drones, are pushing the scientific community to fully exploit the potential of these fascinating robots. Drones have an intrinsic advantage to be able to access environments that are otherwise hard to reach. They are already used in various areas such as safety (surveillance, search and rescue missions), delivery services, cinematography and entertainment (drone racing). For some of these applications, their success hinges on how autonomously and fast drones can operate. In addition, considering the growing number of drones in the sky, it becomes increasingly necessary to develop autonomous guidance and control systems.

A particular area of interest is how to make the drones arrive to their destination in the minimal amount of time possible, especially for drones that do not have a fixed wing (such as quadrotors), as they tend to have a limited flight range [1]. Time-optimal flight is challenging because the drone has to execute very aggressive maneuvers which pushes the platform to the limits of its flight envelope, leaving little room to recover from modelling errors and disturbances. On top of this, as drones fly faster, aerodynamic effects tend to become more significant and harder to model, see [2]. A promising avenue for autonomous guidance and control is the use of neural networks, so-called Guidance and Control Networks (G&CNETs, see [3]), due to their low computational cost during inference and ability to learn the mapping from states to optimal control inputs. This research aims to explore this avenue and push current G&CNETs to be more robust and time optimal.

Unmanned aerial vehicles (UAVs) can be subdivided based on the number of rotors they possess or the presence of a fixed wing. Although fixed wing drones offer an attractive solution for missions that require the vehicle to fly for long periods of time, they lack maneuvrability and vertical take-off and landing is challenging. Quad-, tri- or hexa-copters on the other hand are much more agile [4, 5] than their fixed-wing counterparts. This research will focus on the control of quadcopter, because it is the most common drone used for agile and fast maneuvers.

The literature review (Part II) is subdivided into four main chapters: optimal control theory (Ch. 1), artificial neural networks (Ch. 2), time-optimal guidance & control of drones (Ch. 3) and bridging the reality gap for G&CNET (Ch.4). The two remaining parts of this document present a Gantt chart for this thesis (Part III) and the research question and objective (Part IV).

II

Literature review

1 Optimal Control Theory

Optimal control theory allows to compute the inputs to a dynamical system such that a predefined cost function is optimized over a period of time. It will be the mathematical framework used in this research to solve the problem of time optimal quadrotor flight. Sec. 1.1 describes how optimal control problem (OCPs) will be formulated in the rest of this work, Sec. 1.2 gives an overview of the history of optimal control theory by diving into the Bellman equations, the Hamilton-Jacobi-Bellman (HJB) equations and Pontryagin's Maximum Principle (PMP). Sec. 1.3 lists common method to solve OCPs, namely indirect methods, direct methods and dynamic programming (DP). Finally, Sec. 1.4 proposes two well known closed loop optimal control methods, Linear Quadratic Regulator (LQR) and Model Predictive Control (MPC). A summary of this chapter is provided at the end (Sec. 1.5).

1.1. General optimal control problem formulation

Before diving into the history and intricacies of optimal control theory, let's define a general formulation of an OCP which will be used in the rest of this work. The dynamics of the system are described by a set of differential equations, where $\mathbf{x}$ is the state vector and $\mathbf{u}$ represents the control inputs:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(t), t)$$

Let $\mathbf{x}(t_0)$ and $\mathbf{x}(t_f)$ be the initial and final conditions, respectively. The cost function to maximize (or minimize) is then:

$$J = h(\mathbf{x}(t_f), t_f) + \int_{t_0}^{t_f} l(\mathbf{x}(t), \mathbf{u}(t), t)dt$$

The former term in the cost function $h(\mathbf{x}(t_f), t_f)$ (sometimes referred to as Mayer term) represents the cost of the final states $\mathbf{x}(t_f)$ and the final time t_f. The latter term sums up the cost over time (from t_0 to t_f) and is referred to as the running cost. For a control sequence $\mathbf{u}^*(t)$[1] to be optimal, the cost function J needs to be minimized (or maximized) over $[t_0 \rightarrow t_f]$, respect boundary conditions and possible additional constraints.

1.2. History of optimal control theory

The history of optimal control dates back to 1638, when Galileo posed the so-called "Brachistochrone" problem. The polymath was interested in finding the trajectory between two points, such that a bead attached to a wire between these points would arrive in the smallest amount of time possible under the effects of gravity (without friction) [6]. As it turns out, the Brachistochrone problem can be formulated as an Optimal Control Problem (OCP), where the control $\mathbf{u}(t)$ is the angle between the local horizontal and the velocity vector of the bead, see Fig. 1.1 from [7]. The control input $\mathbf{u}(t)$ can take any value in the set of admissible controls $U_{ad} := \{\mathbf{u} : [0, t_f] \rightarrow (0, 2\pi) :$ u continuous$\}$. Galileo's conclusion on the solution of his problem turned out to be incorrect, it is only half a century later, when Johann Bernoulli

[1] superscript * denotes optimality

challenged other mathematicians that the problem was solved. The challenge from Bernoulli sparked interest in the world of mathematics and eventually led to the development of ”calculus of variations” [6].

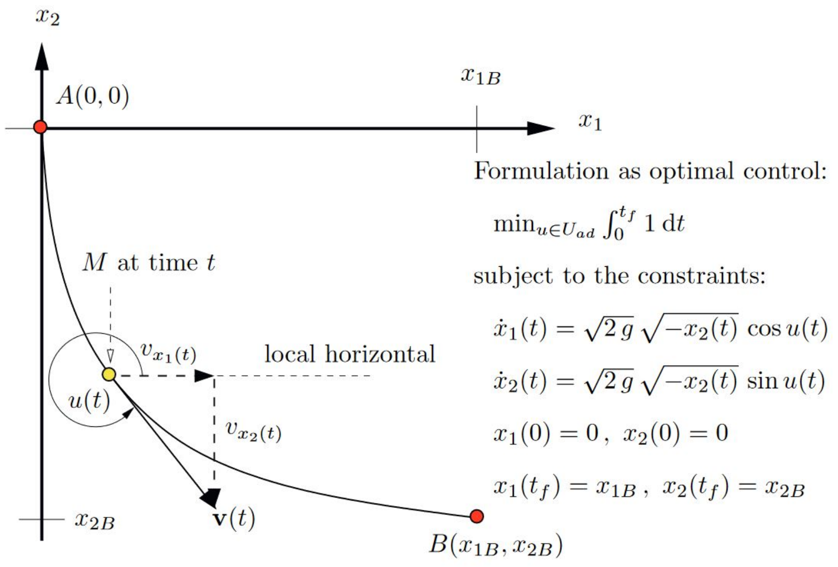

x_2
$A(0,0)$
x_{1B}
x_1
Formulation as optimal control:
$\min_{u \in U_{ad}} \int_0^{t_f} 1 \, \mathrm{d}t$
subject to the constraints:
$\dot{x}_1(t) = \sqrt{2\,g}\,\sqrt{-x_2(t)}\,\cos u(t)$
$\dot{x}_2(t) = \sqrt{2\,g}\,\sqrt{-x_2(t)}\,\sin u(t)$
$x_1(0) = 0\,,\ x_2(0) = 0$
$x_1(t_f) = x_{1B}\,,\ x_2(t_f) = x_{2B}$
M at time t
$v_{x_1}(t)$
local horizontal
$u(t)$
$v_{x_2}(t)$
x_{2B}
$\mathbf{v}(t)$
$B(x_{1B}, x_{2B})$

Figure 1.1: Brachistochrone problem formulated as on OCP by [7]

Fast-forward to the 20th century when advances in computational performance give rise to numerical solvers, Richard Bellman and Lev Pontryagin laid the groundwork for what is known today as optimal control theory. Richard Bellman formulated the principle of optimality [8], which is at the basis of Dynamic Programming (DP). Lev Pontryagin on the other hand developed the first order necessary, but not sufficient conditions for optimality in a principle named after him: Pontryagin’s Maximum (or Minimum) Principle (PMP). The main difference between these two pillars of optimal control theory is that dynamic programming suffers from the curse of dimensionality as the entire state space is searched for an optimal solution. PMP however, avoids this curse and only applies to deterministic problems [9]. In the following, the main mathematical principles that originated from these two mathematicians are described as done by Todorov [9], who summarized these principles concisely: the Bellman equations (Subsec. 1.2.1), the Hamilton-Jacobi-Bellman equations (Subsec. 1.2.2) and Pontryagin’s Maximum Principle (Subsec. 1.2.3).

1.2.1. The Bellman equations

Dynamic Programming (DP) is particularly useful to solve discrete control problems. It stems from Bellman optimality principle, which relies on the fact that:

> ”[...] if a given state-action sequence is optimal, and we were to remove the first state and action, the remaining sequence is also optimal (with the second state of the original sequence now acting as initial state).”[9]

This means that OCP can be solved by starting at the final conditions and finding the optimal state-control actions recursively since optimal controls are independent of past decisions. An important quantity in this context is the so-called optimal value function $v(\mathbf{x})$ (sometimes called cost-to-go function) [9]. This function gives the minimal cost to complete the optimization problem, starting from state $\mathbf{x}$. The optimal value function is used in a lot of optimization methods to determine a control law $\pi(\mathbf{x})$, which maps the states to the corresponding optimal controls: $\pi : \mathcal{X} \rightarrow \mathcal{U}(\mathcal{X})$. Where $\mathcal{X}$ and $\mathcal{U}$ are both finite sets for the states and admissible controls, respectively. Using the notation in [9], let’s define $next(\mathbf{x}, \mathbf{u}) \in \mathcal{X}$ as the state that results from applying the control input $\mathbf{u}$ at state $\mathbf{x}$ and $cost(\mathbf{x}, \mathbf{u}) \geq 0$ as the cost of applying the control input $\mathbf{u}$ at state $\mathbf{x}$. The optimal control law then needs to satisfy:

$$\pi(\mathbf{x}) = \arg \min_{\mathbf{u} \in \mathcal{U}(\mathbf{x})} \{cost(\mathbf{x}, \mathbf{u}) + v(next(\mathbf{x}, \mathbf{u}))\} \tag{1.1}$$

Note that the resulting control law $\pi(\mathbf{x})$ is not necessarily unique, there may be multiple optimal control strategies. The optimal value function needs to satisfy the following equation, for which there is a unique solution:

$$v(\mathbf{x}) = \min_{\mathbf{u}\in\mathcal{U}(\mathbf{x})}\{cost(\mathbf{x},\mathbf{u}) + v(next(\mathbf{x},\mathbf{u}))\} \tag{1.2}$$

Eq.1.1 and 1.2 are the so-called Bellman equations used to solve discrete OCPs. The equations above are used to solve deterministic problems, nevertheless they can be easily extended to stochastic problems, where the possible states that result from applying a certain control action are expressed as a probability distribution. If an OCP has discrete states and actions with stochastic state transitions it is called a Markov Decision Process (MDP) [9].

1.2.2. The Hamilton-Jacobi-Bellman equations

Optimal controls problems can also be solved for continuous cases, where $\mathcal{X}$ and $\mathcal{U}$ are not finite sets. Let's derive the so-called Hamilton-Jacobi-Bellman equations for continuous OCP, following the steps taken in [9]. The state and controls of the system can now take any value in $\mathbf{x} \in \mathbb{R}^{n_x}$ and $\mathbf{u} \in \mathcal{U}(x) \subset \mathbb{R}^{n_u}$, where n_x and n_u are the number of states and control inputs, respectively.

Starting from a stochastic differential equation (using $d\mathbf{w}$ as Brownian motion) of the form [9]:

$$d\mathbf{x} = f(\mathbf{x},\mathbf{u})dt + F(\mathbf{x},\mathbf{u})d\mathbf{w} \tag{1.3}$$

The term $F(\mathbf{x},\mathbf{u})$ represents the noise of the system. Let's take the integral of the Eq. 1.3:

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t f(\mathbf{x}(s),\mathbf{u}(s))ds + \int_0^t F(\mathbf{x}(s),\mathbf{u}(s))d\mathbf{w}(s)$$

The last term can be written as:

$$\int_0^t g(s)d\mathbf{w}(s) = \lim_{n\to\infty}\sum_{k=0}^{n-1} g(s_k)(\mathbf{w}(s_{k+1}) - \mathbf{w}(s_k))$$

where $0 = s_0 < s_2 < \dots < s_n = t$. Let's now apply the following Euler discretization along the time axis:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta f(\mathbf{x}_k,\mathbf{u}_k) + \sqrt{\Delta}F(\mathbf{x}_k,\mathbf{u}_k)\epsilon_k$$

where $\epsilon_k \sim \mathcal{N}(0, I^{n_w})$ (n_w is the dimension of the noise vector) and Δ is the time step such that $\mathbf{x}_k = \mathbf{x}(k\Delta)$. A distinction has to be made between finite and infinite time horizon. Let's consider a typical cost function for a finite time horizon (i.e. t_f is specified):

$$J(\mathbf{x}(\cdot),\mathbf{u}(\cdot)) = h(\mathbf{x}(t_f)) + \int_0^{t_f} l(\mathbf{x}(t),\mathbf{u}(t),t)dt$$

Discretizing the cost function yields:

$$J(\mathbf{x}_.,\mathbf{u}_.) = h(\mathbf{x}_n) + \Delta\sum_{k=0}^{n-1} l(\mathbf{x}_k,\mathbf{u}_k,k\Delta)$$

where n represents the amount of time steps such that $n = \frac{t_f}{\Delta}$. Since this is a stochastic problem, the transition between states is expressed as a multivariate Gaussian probability distribution:

$$\mathbf{x}_{k+1} \sim \mathcal{N}(\mathbf{x}_k + \Delta f(\mathbf{x}_k,\mathbf{u}_k), \Delta S(\mathbf{x}_k,\mathbf{u}_k))$$

where $S(\mathbf{x},\mathbf{u}) = F(\mathbf{x},\mathbf{u})F(\mathbf{x},\mathbf{u})^T$. In this case the optimal value function v depends both on the states and time:

$$v(\mathbf{x},k) = \min_{\mathbf{u}}\{\Delta l(\mathbf{x},\mathbf{u},k\Delta) + E[v(\mathbf{x} + \Delta f(\mathbf{x},\mathbf{u}) + \xi, k+1)]\} \tag{1.4}$$

where $v(\mathbf{x}, n) = h(\mathbf{x})$ and $\xi \sim \mathcal{N}(0, \Delta S(\mathbf{x}, \mathbf{u}))$. Let's expand the optimal value function using a second order Taylor series (the time indices are omitted for brevity):

$$v(\mathbf{x} + \delta) = v(\mathbf{x}) + \delta^T v_{\mathbf{x}}(\mathbf{x}) + \frac{1}{2}\delta^T v_{\mathbf{xx}}\delta + O(\delta^3)$$

where $\delta = \Delta f(\mathbf{x}, \mathbf{u}) + \xi$ and the subscripts $\square_{\mathbf{x}}$ and $\square_{\mathbf{xx}}$ stand for the first and the second partial derivative with respect to $\mathbf{x}$, respectively. Now let's omit the terms higher than the first-order in Δ and take the expectation of the optimal value function:

$$E[v] = v(\mathbf{x}) + \Delta f(\mathbf{x}, \mathbf{u})^T v_{\mathbf{x}}(\mathbf{x}) + \frac{1}{2}tr(\Delta S(\mathbf{x}, \mathbf{u}) v_{\mathbf{xx}}(\mathbf{x})) + O(\Delta^2)$$

where $tr(\Delta S(\mathbf{x}, \mathbf{u}) v_{\mathbf{xx}}(\mathbf{x}))$ stands for the trace of the matrix $\Delta S(\mathbf{x}, \mathbf{u}) v_{\mathbf{xx}}(\mathbf{x})$ (sum of main diagonal elements). Substituting the expectation of the optimal value function into Eq. 1.4, taking $v(\mathbf{x})$ out of the $\min$ operator and dividing by the time step:

$$\frac{v(\mathbf{x}, k) - v(\mathbf{x}, k+1)}{\Delta} = \min_{\mathbf{u}}\{l + f^T v_{\mathbf{x}} + \frac{1}{2}tr(S v_{\mathbf{xx}}) + O(\Delta)\}$$

Since $t = k\Delta$, the expression above can be written in the continuous time domain, the left hand side then becomes:

$$\frac{v(\mathbf{x}, t) - v(\mathbf{x}, t + \Delta)}{\Delta}$$

Taking the limit $\Delta \to 0$ yields the partial derivative $-\frac{\partial}{\partial t}v$ or $-v_t$. Hence over $0 < t < t_f$ and setting $v(\mathbf{x}, t_f) = h(\mathbf{x})$ one gets:

$$-v_t(\mathbf{x}, t) = \min_{\mathbf{u} \in \mathcal{U}(\mathbf{x})}\{l(\mathbf{x}, \mathbf{u}, t) + f(\mathbf{x}, \mathbf{u})^T v_{\mathbf{x}}(\mathbf{x}, t) + \frac{1}{2}tr(S(\mathbf{x}, \mathbf{u}) v_{\mathbf{xx}}(\mathbf{x}, t))\} \tag{1.5}$$

The optimal control law $\pi(\mathbf{x}, t)$ then needs to obey:

$$\pi(\mathbf{x}, t) = \arg\min_{\mathbf{u} \in \mathcal{U}(\mathbf{x})}\{l(\mathbf{x}, \mathbf{u}, t) + f(\mathbf{x}, \mathbf{u})^T v_{\mathbf{x}}(\mathbf{x}, t) + \frac{1}{2}tr(S(\mathbf{x}, \mathbf{u}) v_{\mathbf{xx}}(\mathbf{x}, t))\} \tag{1.6}$$

Eq. 1.5 and 1.6 are the so-called Hamilton-Jacobi-Bellman (HJB) equations for continuous-time stochastic control problems. Bellman notes that the HJB equations suffer from what he calls the "curse of dimensionality". The reason for this is that the only way to guarantee convergence towards the global optimal value function is the use numerical methods that discretize the entire state space. Unfortunately, the number of required discrete states scales exponentially with the state vector dimension n_x. Hence, for complex dynamical systems (such as 12 degrees of freedom quadrotor models) solving the HJB becomes computationally intractable. Fortunately, Bellman's efforts led to other numerical approaches (using parametric models) that yield useful approximate solutions [9].

1.2.3. Pontryagin's Maximum Principle

Turning now to the other pillar of optimal control theory, Pontryagin's Maximum Principle (PMP) [10]. Contrary to Bellman's optimality principle, PMP works only for deterministic systems but does not suffer from the curse of dimensionality. Let's derive PMP starting from the HJB equations. Note that other derivations also exist which use Lagrange multipliers to derive PMP. Following the steps in [9]:

Let's simplify Eq. 1.5 by keeping the deterministic terms:

$$-v_t(\mathbf{x}, t) = \min_{\mathbf{u}}\{l(\mathbf{x}, \mathbf{u}, t) + f(\mathbf{x}, \mathbf{u})^T v_{\mathbf{x}}(\mathbf{x}, t)\} \tag{1.7}$$

Let's define the optimal control law $\pi(\mathbf{x}, t)$ to be a solution to Eq. 1.7 and let it be differentiable in x. Hence $\mathbf{u} = \pi(\mathbf{x}, t)$ such that:

$$0 = v_t(\mathbf{x}, t) + l(\mathbf{x}, \pi(\mathbf{x}, t), t) + f(\mathbf{x}, \pi(\mathbf{x}, t))^T v_{\mathbf{x}}(\mathbf{x}, t)$$

Let's take the partial derivative with respect to $\mathbf{x}$:

$$0 = v_{t\mathbf{x}} + l_{\mathbf{x}} + \pi_{\mathbf{x}}^{t} l_{\mathbf{u}} + (f_{\mathbf{x}}^{T} + \pi_{\mathbf{x}}^{T} f_{\mathbf{u}}^{T}) v_{\mathbf{x}} + v_{\mathbf{xx}} f$$

Simplifying the equation using $\dot{v}_{\mathbf{x}} = v_{\mathbf{xx}}\dot{\mathbf{x}} + v_{t\mathbf{x}} = v_{\mathbf{xx}} f + v_{t\mathbf{x}}$:

$$0 = \dot{v}_{\mathbf{x}} + l_{\mathbf{x}} + f_{\mathbf{x}}^{T} v_{\mathbf{x}} + \pi_{\mathbf{x}}^{T}(l_{\mathbf{u}} + f_{\mathbf{u}}^{T} v_{\mathbf{x}})$$

As it turns out $(l_{\mathbf{u}} + f_{\mathbf{u}}^{T} v_{\mathbf{x}}) = 0$ (assuming this is an unconstrained optimization problem) since this term is the gradient of the term inside the $\min$ operator on the left hand side of Eq. 1.7 with respect to the control input u:

$$-\dot{v}_{\mathbf{x}} = l_{\mathbf{x}}(\mathbf{x}, \pi(\mathbf{x}, t), t) + f_{\mathbf{x}}^{T}(\mathbf{x}, \pi(\mathbf{x}, t)) v_{\mathbf{x}}(\mathbf{x}, t) \tag{1.8}$$

Let's introduce the costate vector $\boldsymbol{\lambda}$ such that $v_x = \boldsymbol{\lambda}$. Pontryagin's Maximum Principle then dictates that if there exists an optimal state-control trajectory $\{x(t), u(t) : 0 \le t \le t_f\}$, then are also exists a trajectory described by a costate vector $\boldsymbol{\lambda}(t)$ such that in Eq. 1.8 v_x can be substituted with $\boldsymbol{\lambda}$ and π with $\mathbf{u}$:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= f(\mathbf{x}(t), \mathbf{u}(t)) \\ -\dot{\boldsymbol{\lambda}}(t) &= l_{\mathbf{x}}(\mathbf{x}(t), \mathbf{u}(t), t) + f_{\mathbf{x}}^{T}(\mathbf{x}(t), \mathbf{u}(t))\boldsymbol{\lambda}(t) \\ \mathbf{u}(t) &= \arg\min_{\underline{\mathbf{u}}}\left\{ l(\mathbf{x}(t), \underline{\mathbf{u}}, t) + f(\mathbf{x}(t), \underline{\mathbf{u}})^{T}\boldsymbol{\lambda}(t)\right\} \end{aligned} \tag{1.9}$$

With $\boldsymbol{\lambda}(t_f) = h_{\mathbf{x}}(\mathbf{x}(t_f))$ and $\mathbf{x}(0)$ as boundary conditions and t_f being given (note that the final time t_f or any final state do not have to be specified in order to solve PMP, so-called transversality conditions allow to leave them unspecified). Looking at Eq. 1.9, one can think of the costate vector $\boldsymbol{\lambda}$ as being the derivative of the optimal value function along an optimal trajectory [9]. PMP is often written in a simplified form using the Hamiltonian:

$$H(\mathbf{x}, \mathbf{u}, \boldsymbol{\lambda}, t) = l(\mathbf{x}, \mathbf{u}, t) + f(\mathbf{x}, \mathbf{u})^{T}\boldsymbol{\lambda} \tag{1.10}$$

The optimal control $\mathbf{u}^{*}(t)$ can then be found by taking the derivative of the Hamiltonian with respect to $\mathbf{u}$ and setting it to zero. The resulting compact form for PMP is:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \frac{\partial}{\partial \boldsymbol{\lambda}} H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) \\ -\dot{\boldsymbol{\lambda}}(t) &= \frac{\partial}{\partial \mathbf{x}} H(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\lambda}(t), t) \\ \mathbf{u}(t) &= \arg\min_{\underline{\mathbf{u}}} H(\mathbf{x}(t), \underline{\mathbf{u}}, \boldsymbol{\lambda}(t), t) \end{aligned} \tag{1.11}$$

The power of PMP lies in the fact that Eq. 1.11 are ordinary differential equations (ODEs). This allows to solve for trajectories without referencing to neighbouring trajectories, which is the case for partial differential equations (PDEs) such as Bellman equations (Eq. 1.5 and 1.6). Eq. 1.11 is treated as a Two Points Boundary Value Problem (TPBVP) which can be solved by standard solvers. Contrarily to Bellman's equations, the computational power required to solve PMP grows linearly with the number states, hence PMP does not suffer from the curse of dimensionality. Note that care must be taken as solutions to the TPBVP which originates from PMP can be local minima. In practice the problem is solved multiple times with different initial guesses to reduce the risk of missing the global optimum.

Eq. 1.11 are the so-called necessary conditions for optimality, which means that if these equations are satisfied the cost function is either maximized or minimized. PMP states that in order to have the necessary and sufficient conditions for optimality the following equation also needs to be satisfied:

$$\mathcal{H}(\mathbf{x}^{*}(t), \mathbf{u}^{*}(t), \boldsymbol{\lambda}^{*}(t)) \le \mathcal{H}(\mathbf{x}^{*}(t), \mathbf{u}(t), \boldsymbol{\lambda}^{*}(t)) \tag{1.12}$$

This inequality makes sure that the solution minimizes the cost function.

1.3. Solving an optimal control problem

Apart from simple linear quadratic problems which can be solved analytically, numerical methods are preferred to solve optimal control problems due to their complexity. According to [11], this is why a lot of efforts in the scientific community have been concentrated on improving such methods since the work of Bellman [12] was published about seven decades ago. One distinguishes between three methods to solve optimal control problems: *indirect* methods, *direct* methods and *dynamic programming*.

Indirect methods stem from the field of mathematics called calculus of variations, see [13], which create a boundary-value problem (BVP). Solutions to this problems are local or global optimal trajectories called extremals. *Direct* methods on the other hand transcribe the optimal control problem to a nonlinear programming problem (NLP) by discretizing either only the controls (*control parameterization*) or both the states and controls (*state and control parameterization*). The problem is subsequently solved using standard optimization techniques. Finally, the last avenue to solve the Hamilton-Jacobi-Bellman (HJB) equations is *Dynamic Programming* (DP), see [9]. This recursive method is useful for low dimensional problems, but suffers from the curse of dimensionality because the entire state space has to be searched to find the optimum. It is hence not suitable for more complex drone models which are needed to formulate time optimal trajectories.

1.3.1. Indirect Methods

To use indirect methods one has to set up the TPBVP resulting from PMP [11]. This branch of optimization (calculus of variations) relies on the first order necessary conditions of optimality (Eq. 1.11) to find candidates for optimal trajectories called *extremals*. Note that such extremals are described by the states and co-states. In some cases, it can be difficult to initialize the solver with an appropriate guess of the co-states, as these variables do not have an intuitive physical meaning. Finally, so-called *transversality conditions* can be added to the TPBVP. These are conditions on the initial or final value of certain co-states that allow to, for instance, leave the boundary conditions of the corresponding states unspecified, which is particularly useful for time-optimal flight, as the minimal total flight time is not known in advance. It also allows to enforce a given relation between different states, an example from trajectory optimization would be to enforce the final magnitude of the velocity to be equal to the circular orbit velocity corresponding to the final orbit radius [14].

The most common indirect methods are the following: single-shooting methods, multiple-shooting methods and collocation methods [11].

Indirect Shooting Method The idea behind a single shooting method is relatively simple. One selects initial conditions for the state and co-states, the system of equations is the integrated in time (from t_0 to t_f) until final conditions for the states and co-states are reached. These are then compared to the desired final boundary conditions. In case this difference is higher than a specified threshold, the solver slightly modifies the initial conditions, propagates the system forward in time again and compares the final states and costates. This process is repeated until the desired final conditions are met. Care should be taken to select appropriate initial conditions, as the solver might diverge or converge to a local minimum instead of the desired global optimum [11]. A popular commercial solver that can be used for such TPBVP is SNOPT (Sparse Nonlinear OPTimizer). A simple pseudo-code algorithm (Fig. 1.2) adapted from [11] is presented below:

```
$\mathbf{x}_0, \boldsymbol{\lambda}_0 \leftarrow$ Input
while Error in final conditions is larger than given threshold do
    Integrate system from $t_0$ to $t_f$
    Compute error between final conditions and desired conditions
    if Error is below given threshold then
        Break
    else
        $\mathbf{x}_0, \boldsymbol{\lambda}_0 \leftarrow$ Update
    end if
end while
```

Figure 1.2: Pseudo-code adapted from [11] describing an indirect single shooting methods

The output of an indirect single shooting method is an extremal, i.e. a trajectory described by its states and co-states.

Indirect Multiple Shooting Method Multiple shooting methods have been developed to overcome some of the numerical difficulties that can arise for single shooting methods [11]. Especially in the case of hyper-sensitive[2] control problems, errors might grow uncontrollably when integrated over the entire time horizon $[t_0, t_f]$. It is then useful to split up the time horizon into smaller intervals $[t_i, t_{i+1}]$, solve each of these intervals with a single shooting method and enforce continuity of the states and co-states between each of these intervals. Fig. 1.3 from [11] depicts the basic principle behind a multiple shooting method, note that the vector $\mathbf{y}(t)$ is a vector containing both the states and the co-states $\mathbf{y}(t) = \begin{bmatrix}\mathbf{x}(t) & \boldsymbol{\lambda}(t)\end{bmatrix}^T$.

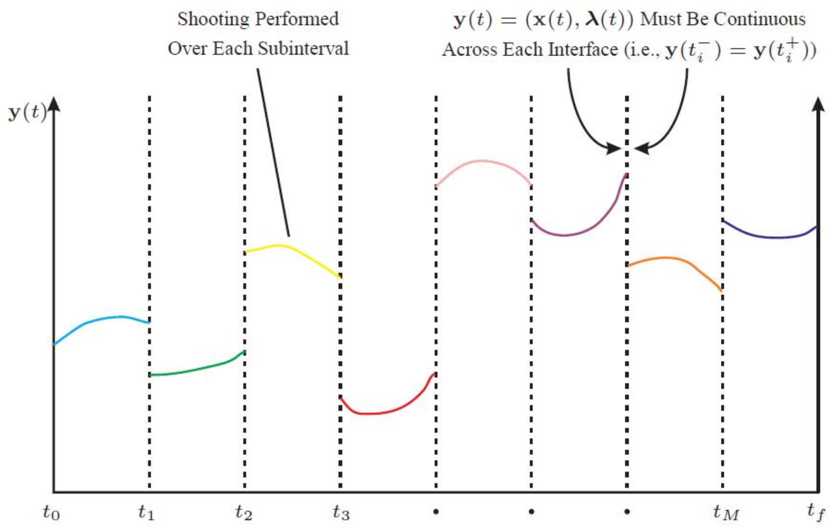

Shooting Performed
Over Each Subinterval
$\mathbf{y}(t) = (\mathbf{x}(t), \boldsymbol{\lambda}(t))$ Must Be Continuous
Across Each Interface (i.e., $\mathbf{y}(t_i^-) = \mathbf{y}(t_i^+)$)
$\mathbf{y}(t)$
t_0
t_1
t_2
t_3
•
•
•
t_M
t_f

Figure 1.3: Figure from [11] illustrating a multiple shooting method

Hence, this boils down to a larger root-finding problem, as the difference between the states and co-states between each sub-interval needs to be driven to zero. Despite the increased amount of variable to solve for, multiple shooting method mitigate some of the numerical instabilities one can encounter with single shooting methods.

Indirect Collocation Method Indirect collocation methods parameterize the states and co-states with polynomials. Applying a certain collocation method results in a root-finding problem where each of these polynomials is changed until the boundary conditions are met. The main difference with direct methods is that for indirect methods the co-states must also be integrated. More details on collocation methods are provided for direct methods as these are more common and more powerful.

1.3.2. Direct Methods

As opposed to indirect methods, direct methods transcribe the OCP to a *nonlinear programming problem* (NLP) and then solve it. This is done by discretizing either only the controls (*control parameterization*) or both the states and controls (*state and control parameterization*). Compared to indirect methods, direct methods provide an approximation of the optimal trajectory.

Direct Shooting Method For direct shooting methods the controls $\mathbf{u}(t)$ need to be parameterized, this can be done using functions $\psi_i(t)$ and coefficients a_i $(i = 1, ..., m)$ such that [11]:

[2]Problems for which the interval of time for the TPBVP is too long compared to the rate at which solutions expand and contract in the neighbourhood of optimal solutions [11]

$$\mathbf{u}(t) \approx \sum_{i=1}^{m} a_i \psi_i(t)$$

The dynamics of the system are then enforced by integrating them forward in time using time-marching algorithms, such as Euler or Runge-Kutta methods [11]. Note that the cost function must be evaluated using an integration scheme that is consistent with the one used to solve the differential equations. The resulting NLP tries to minimize this cost while taking into account any given constraints. The pseudo-code below (Fig. 1.4), adapted from [11], describes a generic single shooting direct method:

```
$\mathbf{a}_i$ ← Input initial guess for parameters
while Cost function is not yet minimized AND Constraints are violated do
    Integrate system from $t_0$ to $t_f$
    Compute error between final conditions and desired conditions
    if Error is below given threshold then
        Break
    else
        $\mathbf{a}_i$ ← Update parameters such that cost function value decreases
    end if
end while
```

Figure 1.4: Pseudo-code adapted from [11] describing a direct single shooting methods

The output of a single shooting direct method are the parameters a_i that minimize the cost function while satisfying the constraints (in case the solver converges).

Direct Multiple Shooting Method The principle behind direct multiple shooting methods is the same as for indirect methods, except that each sub-interval is solved for using the direct single shooting algorithm presented above. The problem also boils down to a root finding problem that tries to enforce continuity between each of the sub-intervals by driving the difference in states to zero.

Direct Collocation Method Direct collocation methods (sometimes called *transcription*) are popular as they address the numerical instabilities of shooting methods. They rely on both state and control parameterization. Based on the type of polynomials used for the parameterization, one distinguishes between two classes of collocation: orthogonal and non-orthogonal. Typical non-orthogonal transcription methods use the Chebyshev or the Legendre polynomials. A non-orthogonal examples is the Hermite-Simpson collocation, which is going to be the preferred method in this work as it is particularly well suited for time-optimal control [15]. All of the aforementioned methods use low order polynomials on time sub-intervals, hence they fall under the category of *local collocation*. There exists another category called *global collocation* that uses higher order polynomials on the entire interval $[t_0, t_f]$ (*pseudospectral* methods).

The Hermite-Simpson collocation method provides multiple advantages which are listed below and is the preferred method in recent work on time-optimal quadrotor flight (MSc thesis Robin Ferede, not published yet). Therefore a more in-depth background is given on this method. One of the advantages of the Hermite-Simpson method is that the system dynamics are approximated by piecewise quadratic functions, which contrary the other collocation methods (such as *trapezoidal* collocation which approximate using linear functions), yields higher-order accurate solutions [16]. In addition, the Hermite-Simpson method allows to express the state trajectory as a cubic spline, which has the advantage that its first order derivative is continuous. Note that the states are discretized at every grid point (collocation point), whereas the controls are only discretized at so-called mid-points.

In time-optimal flight, the cost function representing the OCP will contain an integral term to be minimized. In the Hermite-Simpson method, integrals are approximated according to the Simpson quadrature, which expresses the function that is being integrated as a piecewise quadratic function:

$$\int_{t_0}^{tf} w(\tau) d\tau \approx \sum_{k=0}^{N-1} \frac{h_k}{6} (w_k + 4w_{k+\frac{1}{2}} + w_{k+1})$$

where $w(\tau)$ is the integrand and $h_k = t_{k+1} - t_k$ is the time step. Note that the subscript $k+\frac{1}{2}$ denotes a mid-point. The system dynamics are enforced using so-called *collocation constraints*, which are obtained by writing the system dynamics $\mathbf{f}(\cdot)$ in integral form. Following the steps of [16]:

$$\dot{\mathbf{x}} = \mathbf{f}$$

$$\int_{t_k}^{t_{k+1}} \dot{\mathbf{x}} dt = \int_{t_k}^{t_{k+1}} \mathbf{f} dt$$

This continuous expression is then transcribed by approximating it with the so-called *Simpson quadrature*:

$$\mathbf{x}_{k+1} - \mathbf{x}_k = \frac{1}{6} h_k (\mathbf{f}_k + 4\mathbf{f}_{k+\frac{1}{2}} + \mathbf{f}_{k+1}) \tag{1.13}$$

In order to also enforce the dynamics at the mid-points, an interpolant is used to evaluate the states at the mid-points [16]:

$$\mathbf{x}_{k+\frac{1}{2}} = \frac{1}{2}(\mathbf{x}_k + \mathbf{x}_{k+1}) + \frac{h_k}{8}(\mathbf{f}_k - \mathbf{f}_{k+1}) \tag{1.14}$$

Since Eq. 1.14 depends only on values at the grid-points, it is possible to combine it with Eq. 1.13, which results in the so-called *compressed form*. It is also possible to keep both equations separated by introducing a decision variable representing the states at the mid-points, this formulation is called *separated form*. The latter form is usually preferred when the number of time sub-intervals used is small [16].

Another major advantage of this method is that constraints on the states, controls, path or boundaries can easily be implemented at specified grid points. The solution to this NLP problem yields the values for the states and controls at every collocation point. Such a trajectory can then be interpolated to obtain the values between these points. In case the collocation points are uniformly space out in time, the function for the control over time becomes [16]:

$$\mathbf{u}(t) = \frac{2}{h_k^2}(\tau - \frac{h_k}{2})(\tau - h_k)\mathbf{u}_k - \frac{4}{h_k^2}(\tau)(\tau - h_k)\mathbf{u}_{k+\frac{1}{2}} + \frac{2}{h_k^2}(\tau)(\tau - \frac{h_k}{2})\mathbf{u}_{k+1}$$

where $\tau = t - t_k$ and $t_{k+\frac{1}{2}} = \frac{1}{2}(t_k + t_{k+1})$. Since the system dynamics $\mathbf{f}(\cdot) = \dot{\mathbf{x}}$ are also expressed by quadratic polynomials on each sub-interval, the following expression holds:

$$\mathbf{f}(t) = \dot{\mathbf{x}} = \frac{2}{h_k^2}(\tau - \frac{h_k}{2})(\tau - h_k)\mathbf{f}_k - \frac{4}{h_k^2}(\tau)(\tau - h_k)\mathbf{f}_{k+\frac{1}{2}} + \frac{2}{h_k^2}(\tau)(\tau - \frac{h_k}{2})\mathbf{f}_{k+1}$$

In order to compute the actual states between the collocation points, the expression above can simply be integrated:

$$\mathbf{x}(t) = \int \dot{\mathbf{x}} dt = \int \left[\mathbf{f}_k + (-3\mathbf{f}_k + 4\mathbf{f}_{k+\frac{1}{2}} - \mathbf{f}_{k+1})(\frac{\tau}{h_k}) + (2\mathbf{f}_k - 4\mathbf{f}_{k+\frac{1}{2}} + 2\mathbf{f}_{k+1})(\frac{\tau}{h_k})^2 \right] dt$$

Let's now set as boundary conditions $\mathbf{x}(t_k) = \mathbf{x}_k$, the expression above can then be simplified to:

$$\mathbf{x}(t) = \mathbf{x}_k + \mathbf{f}_k(\frac{\tau}{h_k}) + \frac{1}{2}(-3\mathbf{f}_k + 4\mathbf{f}_{k+\frac{1}{2}} - \mathbf{f}_{k+1})(\frac{\tau}{h_k})^2 + \frac{1}{3}(2\mathbf{f}_k - 4\mathbf{f}_{k+\frac{1}{2}} + 2\mathbf{f}_{k+1})(\frac{\tau}{h_k})^3$$

which can be used to compute the states at any time step.

1.3.3. Dynamic Programming

Solving OCPs in discrete-time using dynamic programming boils down to solving the Bellman equations (Eq.1.1 and 1.2). One can also use dynamic programming for the continuous-time case, for which the HJB equations (Eq. 1.5 and 1.6) need to be solved. Unfortunately, models representing the dynamics of quadrotors are too complex to be solved analytically, which means one needs to rely on numerical methods to search the entire state-space. As discussed before the curse of dimensionality limits such methods to problems with few degrees of freedom. In order to derive time-optimal trajectories, dynamic models considering all the degrees of freedom of a quadrotor need to be used, hence DP is not an ideal candidate for this research.

1.4. Closed loop control

So far the aforementioned control strategies do not take state feedback into account. This section goes over two of the main closed loop optimal control methods, namely linear quadratic regulator (LQR) and model predictive control (MPC).

1.4.1. Linear Quadratic Regulator (LQR)

LQR is a special form of optimal control that it widely used for system with linear dynamics and a quadratic cost function. In case of nonlinear dynamics one needs to linearize the system around the state of interest to apply LQR. Consider a linear time-invariant system described by the following dynamics and initial conditions:

$$\dot{\mathbf{x}} = A(t)\mathbf{x} + B(t)\mathbf{u}$$
$$\mathbf{x}(t_0) = \mathbf{x}_0$$
$$\mathbf{u}(t_0) = \mathbf{u}_0$$

with $\mathbf{x} \in \mathbb{R}^{n_x}$ and $\mathbf{u} \in \mathbb{R}^{n_u}$. The quadratic cost function takes the following general form for an infinite time solution:

$$J(\mathbf{x}, \mathbf{u}) = \int_0^\infty \tilde{\mathbf{x}}^T Q \tilde{\mathbf{x}} + \tilde{\mathbf{u}}^T R \tilde{\mathbf{u}} dt$$

where $\tilde{\mathbf{x}} = \mathbf{x} - \mathbf{x}_0$ and $\tilde{\mathbf{u}} = \mathbf{u} - \mathbf{u}_0$ are the errors and Q and R are positive definite matrices that can be tuned to give more or less weight to the state errors or the control effort, respectively. Following the steps in [17], let's assume that the form of the optimal cost-to-go function is quadratic: $J^*(\mathbf{x}) = \mathbf{x}^T P \mathbf{x}$. One can then solve for P using the Riccati Equation:

$$0 = PA + A^T P - PBR^{-1}B^T P + Q$$

This results in the following optimal control law:

$$\mathbf{u}^* = \mathbf{u}_0 + K(\mathbf{x} - \mathbf{x}_0)$$

using the following gains:

$$K = -R^{-1}B^T P$$

The solution to this problem can be found using dynamic programming [17]. LQR is popular for its robustness and offers a computationally efficient solution for onboard applications. However the equations describing the dynamics of a quadrotor are highly nonlinear and the desired cost function is not necessarily quadratic. Hence LQR is not a suitable approach to the OCP in this research.

1.4.2. Model Predictive Control (MPC)

Model Predictive control on the other hand allows to control linear as-well as nonlinear systems. This branch of control theory emerged during the late seventies and since then it has created a multitude of popular control techniques. The basic idea behind MPC is to use an explicit model of the system to predict future states over a specific time *horizon*. These predictions are then used to generate a control law while optimizing a given cost function [18]. The term *receding horizon* denotes the fact that at each time step, the horizon is shifted forward so that the control input that is fed to the system is always the first one in the sequence of controls that were just computed. Fig. 1.5 from [19] depicts the principle behind the receding time horizon.

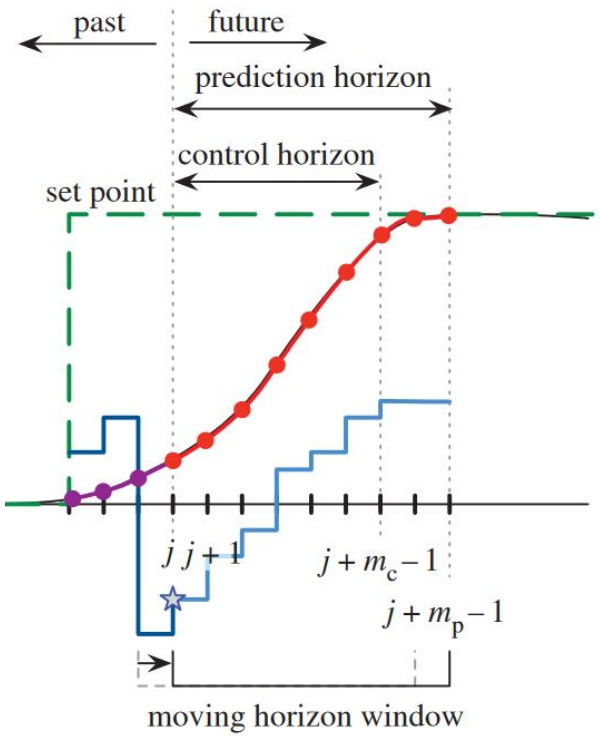


Figure 1.5: Figure from [19] of the receding time horizon for a MPC. The blue line represents the control inputs, the red line the prediction of future states and the green line is the target state. Note that the star represents the control input that is fed to the system.

There exists a multitude of different MPC algorithms: Dynamic matrix control, Model Algorithmic Control, Predictive Functional Control, ... [18]. Depending the model, noise and cost function a certain MPC algorithm might be better suited than another. Following the definition in [19], let's focus on a general formulation of a model predictive controller. The goal is to determine a sequence of control inputs $\mathbf{u}(\cdot|\mathbf{x}_j) := \{\mathbf{u}_{j+1}, ..., \mathbf{u}_{j+k}, ..., \mathbf{u}_{j+m_c}\}$, where $T_c = m_c \Delta t$ denotes the so-called control horizon and $T_p = m_p \Delta t$ is the prediction horizon. The model is sampled using the time step Δt. Note that if the control horizon is shorter than the prediction horizon, it is assumed that the control input is constant after the control horizon. The implicit control law used to solve for the sequence of control inputs $\mathbf{u}(\cdot|\mathbf{x}_j)$ at each timestep is:

$$K(\mathbf{x}_j) = \mathbf{u}(j+1|\mathbf{x}_j) = \mathbf{u}_{j+1}$$

A typical formulation of the cost function being minimized at every timestep is:

$$\min_{\hat{\mathbf{u}}(\cdot|\mathbf{x}_j)} J(\mathbf{x}_j) = \min_{\hat{\mathbf{u}}(\cdot|\mathbf{x}_j)} = \left[||\hat{\mathbf{x}}_{j+m_p} - \mathbf{x}^*_{m_p}||^2_{\mathbf{Q}_{m_p}} + \sum_{k=0}^{m_p-1} ||\hat{\mathbf{x}}_{j+k} - \mathbf{x}^*_k||^2_{\mathbf{Q}} + \sum_{k=1}^{m_c-1} (||\hat{\mathbf{u}}_{j+k}||^2_{\mathbf{R}_u} + ||\Delta\hat{\mathbf{u}}_{j+k}||^2_{\mathbf{R}_{\Delta u}}) \right]$$

where the superscript ^ denotes a prediction and $\mathbf{x}_k$ are the measured states. This cost function accounts for errors in future states predictions compared to the reference trajectory $\mathbf{x}^*_k$ and a terminal cost term $\hat{\mathbf{x}}_{m_p}$. Control effort and high rates of control ($\Delta\mathbf{u}_k = \mathbf{u}_k - \mathbf{u}_{k-1}$) can also be part of the minimization. Note that in this formulation, the terms are computed using weight matrices, so that $||\mathbf{x}||^2_{\mathbf{Q}} := \mathbf{x}^T\mathbf{Q}\mathbf{x}$. $\mathbf{Q}$ and $\mathbf{Q}_{m_p}$ are positive definite matrices whereas $\mathbf{R}_u$ and $\mathbf{R}_{\Delta u}$ are semi-definite. The time horizon problem is subject to the following (discrete-time) dynamics:

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{F}}(\hat{\mathbf{x}}_k, \mathbf{u}_k)$$

and possible constraints on the control inputs:

$$\Delta\mathbf{u}_{\min} \leq \Delta\mathbf{u}_k \leq \Delta\mathbf{u}_{\max}$$
$$\mathbf{u}_{\min} \leq \mathbf{u}_k \leq \mathbf{u}_{\max}$$

To name just a few, MPC is advantageous for its ability to control systems with complex dynamics such as nonminimum phase systems, constraints can easily be implemented and the cost function can be tuned. The efficacy of MPC is highly dependent on how well the model predicts future outcomes. The main downside of MPC is that it is very computationally expensive to run online, which renders it impractical for real-time control of drones [19]. However, one should not completely discredit MPC for quadrotors, as recent advances in computational power and new algorithms make it possible to use it for online applications [20].

1.5. Chapter Summary

Optimal control theory rests on multiple pillars, which depending on the application offer certain advantages. Discrete-time OCPs (deterministic or stochastic) can be solved by formulating the Bellman equations and continuous-time OCPs require the HJB equations. They both rely on the important fact that the optimality of certain states and corresponding action does not rely on prior states and actions. Pontryagin's Maximum Principle only applies to deterministic OCPs but has the advantage that it does not suffer from the curse of dimensionality. The three main avenues to solve OCPs are indirect methods, which create a BVP, direct methods which create a NLP and dynamic programming. When an OCP is too complex to be solved analytically, one can resort to numerical methods, such as DP, which search the entire state space for a global optimum. Unfortunately, the solver time scales badly with the system's dimensionality. In the case of 12 DoF quadrotor models, the problem becomes intractable due to the curse of dimensionality.

For this research, the Hermite Simpson direct collocation method will be used, as it is particularly well suited to solve time-optimal trajectories for quadrotors. Direct methods may only provide an approximation of the true global solution, or sometimes get stuck in local minima, but they tend to converge faster and more often than indirect methods. This is crucial as a lot of trajectories need to be generated to form a useful training dataset for G&CNET (See. 3.2). As has been done in another quadrotor application [3], the AMPL language which helps formulating the OCP and the commercial numerical solver SNOPT will be used to implement the above mentioned method in Python.

2 Artificial Neural Networks

Artificial neural networks (ANNs) denote any architecture of neurons and layers connected to each other . Since the early 1990's ANNs have proven to be powerful to approximate functions or recognize patterns in data [21]. Applications range from images classification, outlier detection, speech recognition to policy search, just to name a few. A particular area of interest for this research is the use of ANNs to control systems. This Chapter aims to give a general overview of the different machine learning paradigms (Sec. 2.1) and of the typical neural net architectures (Sec. 2.2). A summary of this chapter is provided at the end (Sec. 2.3).

2.1. Machine learning paradigms

Machine learning algorithms are used to create a function by *learning* from data. The resulting function or model learns to map the input data to some desired output $f : \mathcal{X} \rightarrow \mathcal{Y}$. Once trained, the model can be used on unseen data [22]. This literature study goes over the three main machine learning paradigms: *supervised* learning, *unsupervised* learning and *reinforcement* learning. All of which make used of data differently to train an ANN.

2.1.1. Supervised learning

In order to train a neural network using supervised learning, the training dataset needs to be organized in pairs of so-called features $x_i \in \mathcal{X}$ and corresponding labels $y_i \in \mathcal{Y}$ (n pairs $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n),\}$ [22]). During training the network learns the mapping between the input space $\mathcal{X}$ and the output space $\mathcal{Y}$, such that when exposed to new data it still outputs the corresponding desired output. Typical examples of supervised learning applications are outliers detection or image classification. However supervised learning can also be used for regression problems in control. In that case the features are the states of the system or any other metric that informs the network enough to predict a continuous control input to steer the system (G&CNET). Supervised learning is at the heart of this research. By solving time-optimal trajectories starting from different initial conditions and all ending at the same final conditions (such as a desired waypoint), a training dataset can be created by discretizing these trajectories. The G&CNET is then trained on these state-action pairs. It learns the mapping of the states to the corresponding optimal control inputs.

2.1.2. Unsupervised learning

Contrary to supervised learning, unsupervised learning does not label the data (y_i). The learning process is then purely used to train the network to recognize patterns in the data. Applications of unsupervised learning range from clustering but also computer vision and language processing. The latter two applications emerged with the rise of *self-supervised* learning. This technique uses labels that are embedded in the features [22], hence one does not need to label the entire dataset manually, which can be a difficult and time-consuming task.

2.1.3. Reinforcement learning

Finally reinforcement learning (RL) is useful for problems that require a sequence of action in order to reach a desired goal. It is also used when the exact solution is not known, RL's trial and error nature can lead to unconventional solutions one would not have thought of in the first place. The learning process involves an *agent* interacting with the *environment*. A *reward function* is then used to inform the agent on how well it performed during an *episode*, based on this information the current policy is updated. Shaping the reward function appropriately is not a trivial task, in fact an entire field of research is dedicated to it. The reward can be sparse (at discrete points in time or space) or continuous. Popular algorithms to train controllers with RL are deep deterministic policy gradient (DDPG), trust region policy optimization (TRPO) and proximal policy optimization (PPO) [23]. Note that in the context of this research, the agent (i.e. the quadrotor) can be trained using RL both in simulation or on the physical platform, both of which have their pros and cons. Training in simulation inevitably results in a reality gap from unmodelled effects while training on the real drone is severely limited in terms of flight time. On top of this, crashes are likely to happen in real life since the agent is exploring the edge of the flight envelope with trial and error.

2.2. Types of artificial neural networks

An ANN is a sequences of neurons connected between layers. There exist a multitude of different architectures which describe how these layers and neurons are connected to each other, this section will go over two of the most common ones: *feedforward neural networks* (FFNNs) and *recurrent neural networks* (RNNs). Before diving into this matter, let's introduce some notation by considering the simplest type of network: the *perceptron*. It consists of only one neuron, takes in multiple inputs and output a single scalar value y, see Fig. 2.1 from [22]. The operation that the perceptron performs can be written as:

$$y = \sigma\left(\sum_{j=1}^{d} w_j x_j + b\right)$$

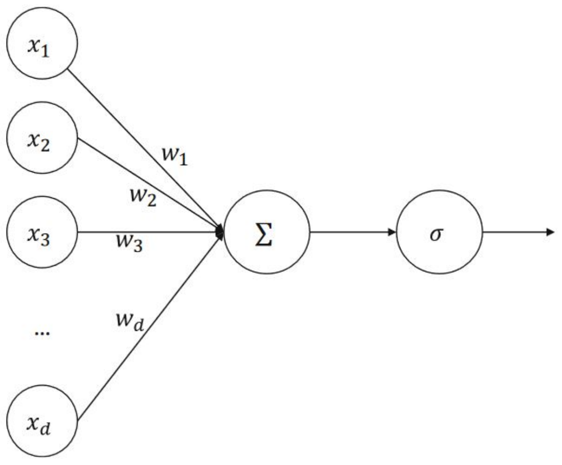


Figure 2.1: Figure from [22] of the perceptron. The activation function is denoted by σ. Note that this figure does not include the (optional) bias b.

where b is an optional bias term and $\sigma(\cdot)$ is the activation function. This function is particularly important as it is the only part that introduces non-linearity (the weights and inputs are linearly summed together). The learning process then consists in finding the weights w_j and bias b such that a specified loss function is minimized. A popular loss function is MSE (*mean squared error*). Examples of popular activation functions are for instance the logistic sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

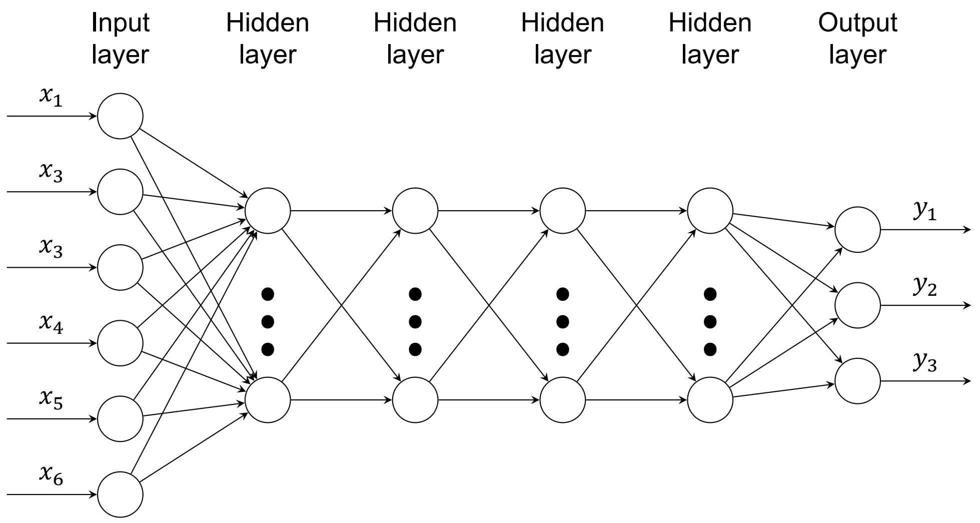


Figure 2.2: Feed forward neural network architecture

and the hyperbolic tangent (TanH):

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Note that there exist countless of different ways to create a ANN: number of neurons, number of hidden layers, scaling of input data, type of activation function, just to name a few. The same goes for neural network training, the type of loss function, number of training epochs (number of times one goes through the entire training data), batch size, learning rate and scheduler are all parameters that will affect the final performance of the ANN. Unfortunately there is no systematic way to tell in advance what setup is best suited for a given application, often one will result to trial and error. It is crucial to try out different network structures as for machine learning the devil is in the details. Another determining factor in the performance of an ANN is the dataset, which is usually split in a training and validation dataset. Usually the larger the dataset and the more representative the data is, the better the final model will perform.

2.2.1. Feedforward neural network

The simplest way to connect neurons to each other is in a feedforward manner. Feedforward neural networks (FFNN) form an acyclic graph as the data passes through it in only one direction [22]: first through the input layer, then through the hidden layers and finally through the output layer, see Fig. 2.2 which contains 6 inputs, 4 hidden layers and one output layer. When neural networks are fully connected between each neurons in each layer they are sometimes called *multilayer perceptrons*. The most common algorithm to train a FFNN is the first order backpropagation algorithm. Another option one can opt for the second order Levenberg-Marquardt which helps to improve the rate of convergence, although this will result in longer training times. The backpropagation algorithm computes the gradient of the cost function J with respect to the weights (and biases if included). The weights are then adapted using this update rule:

$$w_{t+1} = w_t - \eta \frac{\partial J}{\partial w_t}$$

where η is the learning rate, which has to be carefully chosen, such as to not get stuck in local minima or make the algorithm diverge. Care should also be taken to choose an appropriate network architecture and learning procedure such that *overfitting* is avoided. This can for instance happen when more neurons are allocated than necessary, resulting in a model that will perform really well on the training dataset but terribly on the validation dataset due to loss of generality.

2.2.2. Recurrent neural network

Contrary to FFNN, RNNs possess feedback connections on top of feedforward ones. The output of a RNN can hence be used as one of the inputs to the network or as input to one of the previous hidden layers. This has the advantage that the network has access to its own history, which might be needed when the network output depends on a specific sequence of inputs.

2.3. Chapter Summary

This chapter gave a really brief introduction into common machine learning paradigms (supervised learning, unsupervised learning and reinforcement learning) and the two main types of neural network architectures (feedforward and recurrent neural networks). One should note the fundamental differences between the different learning paradigms, supervised learning is used to learn to map the features (such as a quadrotor's states) to given labels (such as the corresponding optimal control inputs). Unsupervised learning on the other hand does not use labels which allows the network to identify patterns in the data by itself. Finally, an analogy to reinforcement learning is how humans learn, namely with trial and error. Contrary to supervised learning, RL creates feedback mechanisms by interacting directly with the environment (either in simulation or on the physical platform). Two main neural network architectures will be of interest in this research, depending on whether only the current states are given as inputs (feedforward neural network) or a sequence of inputs (recurrent neural network).

Contrary to other applications of machine learning, such as computer vision, using flight data to improve G&CNET is not as straightforward. For instance, if a neural network is used to detect obstacles around the quadrotor, one can easily close the reality gap by retraining the network with the obstacles it didn't detect. In that case the obstacles would be manually labelled using flight data (i.e. the images). This systematic approach is very powerful, however it is not clear yet how flight data can be used in a similar fashion to improve G&CNET. Ch. 4 will delve deeper into the reality gap for G&CNETs.

3 Time-optimal Guidance & Control of Drones

Time-optimal guidance and control for autonomous drones has become a central point of attention in recent years [1, 24, 25]. Drone racing competitions like the AlphaPilot challenge [26] or AgileFlight [25] are organized to push research towards more autonomous and faster drones. Researches have tried to tackle this optimal control problem from different angles. One often distinguishes between two types of control strategies: trajectory tracking methods (Sec. 3.1) and trajectory optimization methods (Sec. 3.2). The former solves the optimal trajectory and then tracks the pre-computed trajectory with a controller, whereas the latter combines both steps of generating the trajectory and the corresponding control inputs together. Although trajectory generation methods are at the center of this research, an overview of trajectory tracking methods is also provided hereby. A summary of this chapter is provided at the end (Sec. 3.3).

3.1. Trajectory tracking methods

A main point of concern in the field of time-optimal control is that in order for drones to be fully autonomous, the actuator controls need to be computed onboard of the drone (sometimes referred to as *online*). For trajectory tracking methods it is usually the first step (trajectory generation) that is too computationally expensive to be run online, whereas the control to track a pre-computed trajectory can be generated online [25]. In order for trajectories to be close to the true time-optimal, one needs to at least take into account the full set of equations of motion and thrust constraints for each actuator. Current state-of-the-art solvers take minutes to hours to solve such a time allocation problem [24]. Not being able to quickly compute the required control inputs has detrimental effects on the robustness of the platform, as small disturbances cannot be corrected for, leading to sub-optimal trajectories or in the worst case a crash. Also note that a truly time-optimal trajectory is at the boundary of the flight envelope of the quadrotor, hence at any point in time, at least one of the four rotors will be at its physical limit, leaving no room for control authority. Even small modelling errors can lead to crashes just because the platform lacks control authority to correct its attitude. However modelling errors are bound to occur, since a single crash can change the aerodynamic proprieties off a drone, hence one needs to make the platform robust by either replanning the trajectory online or limiting the actuation limits [25].

There is a trade-off between using the full dynamical model, which makes the time allocation problem more complex, and using a simpler dynamical model which could be solved online. Computationally efficient algorithms which can be run online usually model the trajectories as polynomials or model the drone as a point mass. Point mass models are fast to solve because the trajectory generation problem has a closed form solution [25]. Researchers managed to solve time-optimal trajectories through multiple waypoints [26], however the point mass simplification cannot take into account rotations in 3 dimensions which are essential as the quadrotor needs to be rotated to point the thrust vector in the correct direction. Hence the resulting trajectories are sub-optimal and cannot by themselves result in time-optimal flight.

The alternative is to use polynomials to generate trajectories. A popular approach for aggressive flight is to exploit the differential flatness property of the dynamics of quadrotors. This property allows to generate smooth trajectories that the under-actuated drone is able to track using robust nonlinear controllers, as done by [27]. These smooth trajectories consist waypoints defined by their position and yaw angle and are connected by polynomials. The optimizer subsequently optimizes the trajectory to make it more time optimal. This method can be run onboard of the drone, but the trajectory generation can lead to infeasible paths since the optimizer does not take the system's dynamics into account. On top of this the full actuator potential is not exploited due to the smoothness of the generated trajectories, see [1], which leaves room for improvements towards more time optimal maneuvers. The smoothness of the control inputs stems from the fact that they are sampled using their derivatives. However, the control profile for time optimal flight usually resembles bang-bang control, which consists of step inputs that allow aggressive maneuvers.

[1] have managed to circumvent the issue of reduced actuator potential in smooth trajectories. In order to optimize a trajectory using numerical methods (i.e. make it time optimal), one needs to associate a cost to each waypoint. This is problematic as one does not know in advance what time to allocate to each waypoint. The authors present a solution that allows to optimize the trajectory and the time optimality simultaneously by introducing two variables that complement each other: a progress variable and a variable representing the proximity to the next waypoint. While the authors succeeded in generating aggressive maneuvers, even beating professional human pilots in terms of lap consistency and lap times, the method cannot be run online as it is computationally expensive and requires a model predictive controller to track the generated trajectory.

Model predictive control (MPC) is a common control strategy for drones. It has the advantage that it can take into account actuator constraints while simultaneously optimizing an objective function, as explained by [2]. Unfortunately, MPC suffers from high computational requirements and the necessity to have an accurate dynamical model. The latter issue was addressed in [2] by making use of neural networks to model the complex aerodynamic effects of the drone using a distributed integration scheme. Not only are neural networks well suited to capture the nonlinearities of the drone's dynamics, since it also has a low computational cost during inference, it doesn't exacerbate the expensive computational requirement that MPC controllers have. The authors tested their framework on a real drone, but the reference signals had to be computed offline and sent to the drone, which were then tracked by a Proportional–Integral–Derivative (PID) controller.

Note that one should not discredit the usefulness of point mass models as they have been used successfully to quickly generate a reference trajectory which is then tracked by a Model Predictive Contouring Controller (MPCC) [25]. This state-of-the-art control method allows to replan the trajectory in real-time, which makes it robust to model mismatches and external disturbances. The MPCC in [25] separates the time-optimal objective from the trajectory generation part, which only consists of a 3-dimensional path generated by a point mass model. The controller then produces the inputs such that the generated path is completed in the minimal amount of time and the position error to the path is minimized. Only considering the full equations of motion and actuator constraints during control optimization allows to implement a computationally efficient algorithm. Note that the optimization problem also uses a receding horizon which further decreases the computational load. Since the trajectories generated by a point mass model can be physically infeasible, the authors use a contouring weight, which tells the optimizer how closely it needs to follow the reference trajectory. This contouring weight is dynamically changed when the quadrotor passes through a gate, as this is the most critical part, the weight is then relaxed for the segments between the gates. Unfortunately, the algorithm is still to computationally expensive to be fully run onboard of a quadrotor, as the real-life flight tests had to result to an offboard computer to generated the control inputs.

From the research cited above, it can seem like researchers are counting on more powerful hardware in the future to solve the algorithms that cannot be run onboard today. However, as transistors become smaller and smaller, it is likely that one will reach the point where Moore's law, which states that:

> "[...] the speed of computers, as measured by the number of transistors that can be placed on a single chip, will double every year or two [...]" [28]

won't hold anymore. With this in mind, the most promising avenue is to generate algorithms that are computationally lighter. In [29], the authors treated the onboard computability as a strict requirement. The guidance and control algorithm is very similar to [25], where the controller is able to track non-feasible trajectories generated by a point mass model. The authors cut down on the amount of calculations required to find the minimal time trajectory using Dijkstra's algorithm. Solving this problem requires to find combinations of velocities along all three axis such that the resulting trajectory is time optimal. To do so, velocities are sampled in a velocity graph. Unfortunately, Dijkstra's algorithm scales quadratically with the amount of edges on the velocity graph, which is why the authors sample the velocities uniformly (instead of randomly) to minimize the amount of edges. This efficient process is then repeated by refocusing a smaller cone of sampled velocities around the previously found trajectory. The algorithm is stopped when the ratio of the time between two trajectories is less than some threshold. This algorithm can be solved in $3.48ms$ on a desktop and in $13.3ms$ on a Jetson TX2, which is a powerful embedded computer that can be implemented on a drone. The computational requirement is reduced even more by only replanning the trajectory every second control iteration. The resulting adaptive algorithm is tested in real life by moving a gate during the flight and creating wind disturbances. This constitutes the state-of-the-art in terms of onboard, adaptive and time-optimal quadrotor flight. The only limitation that the authors mention is the fact that the state estimation is done by an offboard motion capture system [29]. Note that the state estimation problem is just as important as the guidance and control problem for time-optimal flight, however it is usually treated as a separate problem.

3.2. Trajectory optimization methods

Trajectory optimization methods treat the two problems of generating optimal trajectories and the corresponding optimal controls as a single block. While directly solving OCPs as introduced in Ch. 1 cannot be done onboard of drones, advances in machine learning such as [30] have shown that neural networks are able to learn the mapping between the current states and the corresponding optimal controls. Essentially learning the solution to the Hamilton–Jacobi–Bellman equations (Eq. 1.5 and 1.6). Since then, the use of neural networks as guidance and control networks (G&CNET) has been demonstrated numerous times, most notably in the field of interplanetary trajectory optimization, see [31–34], but also for quadrotors, see [3]. Another major advantage of neural nets is that they require low computational power during inference, which is a crucial factor for quadrotors because of their limited onboard capacity. The main downsides of such approaches is the time-consuming training process which usually requires a large amount of training data. In addition, neural networks are not well suited to be tuned for a specific platform after training. While these methods have their limitations, they are a promising approach as the entire control pipeline is informed about the quadrotor dynamics, current states and possible constraints.

Reinforcement learning is a popular approach, similar to how a human pilot would learn to fly a drone with trial and error, RL has the power to inform the network on what states or attitudes are desired and which ones should be avoided (such as flying too close to a gate). This fact is leveraged in [35] by training an adaptive model using reinforcement learning, that replans trajectories online at a low computational cost. For each training episode a new random track is generated such that the agent is exposed to a large variety of maneuvers. The OCP is formulated as an infinite-horizon Markov Decision Process (MPD). The RL process then tries to find the optimal control policy such that a progress and safety reward is optimized. Since the total time of flight is not known in advance, the progress reward is formulated by projecting the quadrotor path onto a straight line that connects the centers of the previous and next gate. A safety reward is then used to encourage the drone to fly through the middle of the gate. Note that this results in suboptimal trajectories, as time-optimal flight often requires the quadrotor to fly right at the apex of the gates. However, there is a trade-off between time optimal flight and finishing the race without a crash, hence informing the agent on the riskiness of flying close to the borders of a gate is a sensible thing to implement. In addition, since RL tends to suffer from long training times, the authors adapt the difficulty of the randomly generated tracks based on how well the agent is performing to maximize the amount of successful episodes. Using the relative gate observation to represent the state of the drone as inputs, the network outputs the corresponding thrust commands. While this method works really well in simulation, the authors only validated the generated trajectories by performing a real test flight on a deterministic track, where a MPC was used to track the generated trajectory.

The authors of [36] leveraged not only the advantages of supervised (imitation) learning to learn from optimal trajectories, but also refined the G&CNET by training it using model-free reinforcement learning. The G&CNET is initialized using a supervised learning scheme. The training data consists of optimal trajectories obtained by solving the discretized HJB equations for a simplified dynamic model since the full 12 degrees of freedom model is intractable (curse of dimensionality). These optimal trajectories for a lower order drone model speed up the RL process as the agent does not have to start learning from scratch. On top of this, the reinforcement learning step can be performed using the full, higher order drone model, which is crucial to robustify the controller. In order to account for observation errors, which are bound to happen on the physical platform, the authors model the observations in the RL environment as a Brownian stochastic process. The resulting neural controller takes the drone's states as inputs and outputs the optimal thrust vector. Subsequent low level controllers then compute the corresponding rotor commands. Unfortunately the authors only verified their controller in simulation, but not on a real drone yet.

Prior work on G&CNETs also focused on training only using supervised learning [3]. The training dataset consists of 250,000 trajectories that are partially time- and power-optimal. For this the cost function of the OCP is defined as follows:

$$J(\epsilon, t_f, \mathbf{u}(t)) = (1-\epsilon)t_f + \epsilon \int_0^{t_f} (u_1(t)^2 + u_2(t)^2)dt \tag{3.1}$$

eq. 3.1 contains a continuation parameter ϵ which weighs the two objectives. This approach is useful is purely time-optimal trajectories ($\epsilon = 0$) lead to very aggressive maneuvers, making the platform more prone to crashes. The power-optimal case ($\epsilon = 1$) leads to smoother control inputs while still resulting in quick lap times (as minimizing t_f also minimizes the total power). Two G&CNETs are trained, one with $\epsilon = 0.5$ and one with $\epsilon = 0.2$. Note that a reduced order (2-dimensional) drone model is used to represent the dynamics, u_1 and u_2 correspond to the left and right rotor control inputs, respectively. The resulting OCP is solved using the Hermite-Simpson transcription (Subsec. 1.3.2). The AMPL language is used to formulate the OCP as it allows to inform the numerical solver (SNOPT - Sparse Nonlinear OPTimizer) on the gradients and Hessian of the problem, which is advantageous as such nonlinear problems tend to be difficult to solve, even with powerful commercial solvers. Each node in the resulting trajectories are then added as a state-action pair to the training dataset, where the states are the features and the action is the label. A simple network architecture consisting of only 3 layers with 100 neurons each suffices to learn the mapping between states and control action. Note that *softplus* activation functions for the hidden layers are particularly well-suited for such applications as they allow to obtain a continuous representation of the control inputs [34], which prevents infeasible and discontinuous jumps in the resulting control profile. The authors of [3] also added a delay to account for the fact that on the real quadrotor, it takes some time between observing the current states, computing the control inputs and sending these inputs to the rotors. Even once the controller has the desired control inputs, it takes some time to reach the new commanded RPMs. Note that this G&CNET learns to map the states to the optimal thrust and pitch acceleration, hence it still requires a lower level controller (Incremental Nonlinear Dynamic Inversion (INDI)) to compute the corresponding rotor inputs. This low level controller is indispensable in this case as the G&CNET was only trained on trajectories generated by a 2-dimensional model. The INDI makes sure during the flight test that the drone keeps flying in a straight line and keeps its yaw angle at $\Psi = 0°$. The flight tests showed promising results, outperforming a controller based on the differential flatness property in terms of quadrotor speed. Since [3] has been published, multiple MSc students from TU Delft have worked on improving this work. Rohan Camlesh Chotalal extended the work to a 6-DoF dynamic model (MSc thesis not published at this time) and Robin Ferede worked on reducing the reality gap and performing consistent, power-optimal laps (MSc thesis not published at this time).

3.3. Chapter Summary

The fundamental difference between trajectory tracking methods and trajectory optimization methods is that the former treats the guidance and control problems separately, whereas the latter solves both problems in one step. In order to make drones fully autonomous, these algorithms will have to be computationally light, such that they can be run onboard. Usually, the trajectory generation problem is the most computationally expensive to solve. Most trajectory tracking methods resort to reduced order, or even point mass models to generate time-optimal trajectories, as solving the full equations of motions OCP can take minutes to hours. The first attempts at time-optimal flight relied on the generation of polynomials by exploiting the differential flatness property of quadrotors. This method is computationally light, but does not allow to generate the time-optimal bang-bang control due to the smoothness of the control inputs. MPC and MPCC are very popular in literature as they allow to take the actuator constraints into account while solving the optimization problem simultaneously. While most work rely on an external computer to generate the control inputs with a MPC, one paper managed to make the MPCC algorithm light enough such that it can be run onboard [29]. Trajectory optimization methods are less common in literature, mainly due to the fact that it is difficult to tune such algorithms and the training of neural networks is very time-consuming. Within trajectory optimization methods, RL is the prevalent choice to train quadrotors, whereas supervised learning approaches remain limited. Given the promising results in [3] and recent MSc thesis work by Rohan Camlesh Chotalal and Robin Ferede, the use of G&CNET will further be explored in this work.

4 Bridging the Reality Gap for G&CNET

The reality gap is term used to define the differences between the simulated environment and real life. This gap exists in all control applications for quadrotors, however in this chapter only the reality gap for G&CNET is considered. It stems mainly from unmodelled dynamics (or inaccurate models) (Sec. 4.1) and the cost function used to generate optimal trajectories (Sec. 4.2). Sec. 4.3 lays out another approach to bridge the reality gap without changing the dynamic model or the cost function, namely abstraction. A summary of this chapter is provided at the end (Sec. 4.4).

4.1. Dynamics model & state estimation

The models used in theory or in simulation do not capture all the dynamics that a real drone encounters, they are simplifications of the complex real-life phenomena. System noise (model inaccuracies) and sensor noise (external disturbances) are present and cause the system to behave differently in real-life than in simulation, leading to a suboptimal trajectory or a crash. Note that given the computational power currently available, it is unrealistic to aim for a perfect quadrotor model or perfect state estimation. The mismatch due to this reality gap is bound to happen and cause errors relative to the theoretical optimal trajectory. Hence, the goal should be to have a dynamical model and a state estimation step that are ”good enough” such that the system has enough time to correct these errors.

One option to make a G&CNET more robust against the noisy and biased inputs from sensors is to train the network on noisy input data using reinforcement learning. This has already been done in [36]. In order to account for the uncertainty in states estimates during the network training, the authors used a differential operator proposed by [37]. This operator models the observations (in this case the quadrotor velocity) as a Brownian stochastic process which is meant to resemble the observations that the controller will encounter in real life, hence making it more robust to these uncertainties. Since this research focuses largely on the guidance and control aspect of quadrotors, the state estimation step will be taking care off by a motion capture system called Opti-Track, as is done in [3]. The measurements from Opti-Track in combination with possible Kalman filters offer state estimation that is more accurate than onboard sensors could offer. Hence, it is not expected that the state estimation step will be the limiting factor towards time-optimal flight. Note however that onboard state estimation is still an active field of research, which is crucial to eventually deploy fully autonomous drones.

For this research the model inaccuracies are likely going to have a detrimental effect on the sim-to-real transfer for time-optimal flight. An option to bridge this gap is to slightly vary the model at each episode (for reinforcement learning) or for each generated optimal trajectory (for supervised learning). This method is called *domain randomization* and has been used to train a network to recognize gates under different lightning conditions [38], but also in the context of G&CNET. Robin Ferede has shown that exposing the G&CNET to slightly different external moment disturbances during training facilitates the transfer to real life. The resulting G&CNET can then use onboard sensor data from the inertial measurement unit (IMU) to be informed on external moment disturbances (MSc thesis and corresponding scientific paper not published yet).

Finally, an obvious solution is to make the already existing model more accurate, i.e. perform system identification on the quadrotor. It is known that not all processes are well modelled, for instance the ramp-up behaviour of actuators on the drone is often modelled using a simple linear model which might differ drastically from the real life behaviour. In addition, quadrotors tend to pitch up when their forward velocity increases, a behaviour that is not well modelled either. While improving such models will likely improve the resulting behaviour of the G&CNET, it is not a good systematic solution as a single crash can change the aerodynamic properties of the drone. In addition, this method does not transfer well when one wants to use a completely different drone.

4.2. Cost function

In the context of optimal control, an inaccurate system model is not the only contributor to the reality gap, the cost function used to generate optimal trajectories has to be chosen really carefully. Note that if one had a ”perfect model”, it would be already possible to fly the quadrotor in the same way as in simulation. The reason the cost function is considered in the reality gap is that it might not fully describe the goal of the application. Purely minimizing time might not be the sole interest of the robot when deployed in real life, robustness also has to be taken into account. Similar to a human pilot, the decision he or she makes will be based on a combination of factors. This decision process is difficult to write down mathematically in a cost function. Even if it were possible to come up with a ”human-like” cost function, one can still run into convergence problems when solving the corresponding OCP. An optimal control strategy might also change during flight, an obvious solution would be to train the G&CNET on different training datasets which all consist of trajectories generated with different cost functions. Unfortunately, unless one informs the network, the G&CNET will likely output an average-out control input, which does not resemble either of the true optimal inputs. This problem can be illustrated by so-called bifurcation points. Imagine a point in front of an obstacle where avoiding the obstacle from the left or from the right both constitute optimal trajectories. A G&CNET trained on both trajectories might output the average control input, which would result in a straight (suboptimal) trajectory into the obstacle.

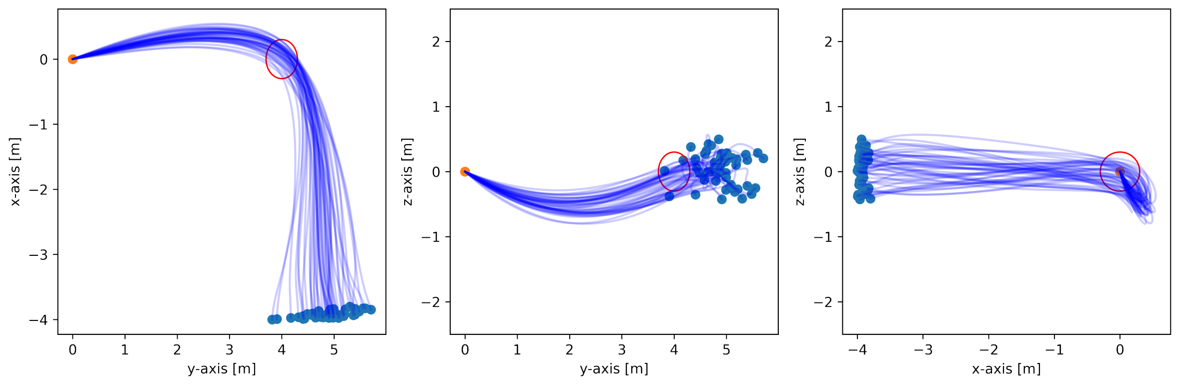


Figure 4.1: Example of 48 optimal trajectories generated using AMPL and SNOPT. The trajectories are required to traverse two consecutive waypoints (the center of red circle and the orange dot, respectively).

A possible avenue which will be explored is to train the G&CNET on two consecutive waypoints instead of just the upcoming one. Coming back to the human pilot analogy, if the pilot knows the relative position of two consecutive gates, he or she will pass through the first gate taking the second gate’s position and orientation into account. It is also possible to only train the G&CNET on the next waypoint, right before reaching that waypoint, one needs to switch to the next waypoint, otherwise the quadrotor will fly into a region of space that does not contain trajectories from the training dataset. This will result in the asymptotic behaviour of the G&CNET, while this is an interesting field of research, it is very difficult to guarantee the stability of the controller outside of the training data. Fig. 4.1 shows a small bundle (collection of trajectories) which could be used to train a G&CNET. The blue dots are the randomly sampled initial conditions and the orange dot is the final position (second waypoint). In this case a constraint was added to the optimization problem which requires the minimal Euclidean distance between the quadrotor and the first waypoint (center of red circle) to be below some variable threshold, here the threshold varies between $15cm$ and $30cm$. Note that one can also opt for a fixed threshold, however there is a risk that the bundle of trajectories becomes very thin at the first waypoint since every trajectory passes through the apex of the gate in order to minimize time. The thinner the bundle of trajectories is, the more likely it is that the quadrotor ends up in states that are outside of the training data. Most likely a trade-off has to be struck between providing the G&CNET with the theoretical optimal trajectories (which is a very difficult OCP for the G&CNET to follow) and suboptimal trajectories (which is an easier OCP to track).

It is also possible to improve the cost function with coach-supervised learning. One can perform multiple flight tests with different cost functions and see which cost function performs best in terms of robustness and speed on certain parts of the track. An adaptive cost function could then be formulated which alters the objective based on the quadrotor's states. This might also give insight into what aspects of the OCP can be relaxed. In theory, the more feasible the OCP is, the easier it will be for G&CNET to guide the drone while satisfying the constraints (and cost function) of this OCP.

4.3. Other approach

A final method to bridge the reality gap in the field of control is abstraction. The idea behind abstraction is train the network on higher-level observations, which are less environment or platform dependent than the raw sensor data. Classical controllers can then be used in combination with the neural net for the low level control. [39] have shown that abstraction can be a solution to more robust neural controllers. The authors used evolutionary algorithms (EAs) to train three micro air vehicles to form a triangle. The transfer from simulation to reality worked for the controller that was trained using abstraction. Another example of abstraction is in [35], where for the real life deployment of the quadrotor the authors only used the neural network to output the optimal trajectory, this trajectory is then tracked using a classical MPC. However, such a network would not be a Guidance & Control Network, since it only takes care of the guidance part. This research solely focuses on making G&CNET more time-optimal, hence abstraction will not be used to bridge the reality gap here.

4.4. Chapter Summary

In the context of G&CNET, the two main contributors to what is called the reality gap, i.e. the difference between the simulated environment and real life, are model mismatches and inadequate cost functions. Striving for a perfect dynamic model is not realistic as aerodynamic properties tend to change over time and due to crashes. A more promising approach is to make the G&CNET robust against model mismatches by training it on trajectories that are generated using a model which contains randomly sampled external disturbances, a process called *domain randomization*. The cost function used to solve the optimal trajectories determines the behaviour of the G&CNET. It will influence how aggressive the maneuvers of the quadrotor are, how robust it is, how feasible the corresponding OCP is and the convergence rate of the numerical solver. A promising approach is to formulate the OCP such that the trajectories pass through two consecutive waypoints, as this informs the network on a larger portion of the entire track.

It is not clear yet whether model mismatches or inappropriate cost functions are the limiting factor when trying to fly quadrotors time-optimally. This is the subject of the preliminary research of this work. Ever more time-optimal G&CNETs will be created by incrementally reducing the continuation parameter ϵ in Eq. 3.1. The resulting networks will then be evaluated on the Parrot Bebop 1 drone (Fig. 4.2) to identify the cause of crashes and guide this thesis.

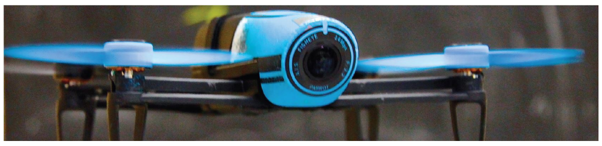
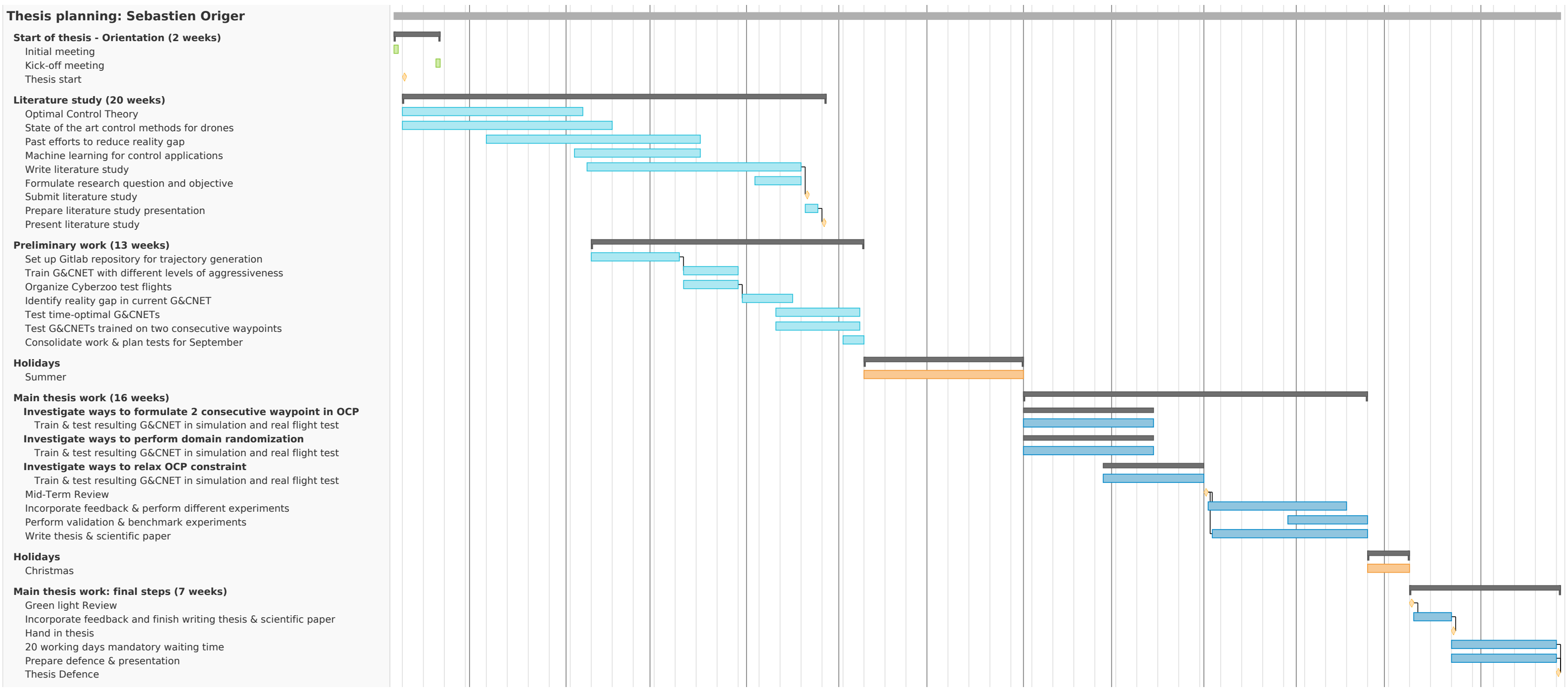


Figure 4.2: Figure from [3] showing the Parrot Bebop 1 drone.

III

Thesis planning


Thesis planning: Sebastien Origer
Start of thesis - Orientation (2 weeks)
Initial meeting
Kick-off meeting
Thesis start
Literature study (20 weeks)
Optimal Control Theory
State of the art control methods for drones
Past efforts to reduce reality gap
Machine learning for control applications
Write literature study
Formulate research question and objective
Submit literature study
Prepare literature study presentation
Present literature study
Preliminary work (13 weeks)
Set up Gitlab repository for trajectory generation
Train G&CNET with different levels of aggressiveness
Organize Cyberzoo test flights
Identify reality gap in current G&CNET
Test time-optimal G&CNETs
Test G&CNETs trained on two consecutive waypoints
Consolidate work & plan tests for September
Holidays
Summer
Main thesis work (16 weeks)
Investigate ways to formulate 2 consecutive waypoint in OCP
Train & test resulting G&CNET in simulation and real flight test
Investigate ways to perform domain randomization
Train & test resulting G&CNET in simulation and real flight test
Investigate ways to relax OCP constraint
Train & test resulting G&CNET in simulation and real flight test
Mid-Term Review
Incorporate feedback & perform different experiments
Perform validation & benchmark experiments
Write thesis & scientific paper
Holidays
Christmas
Main thesis work: final steps (7 weeks)
Green light Review
Incorporate feedback and finish writing thesis & scientific paper
Hand in thesis
20 working days mandatory waiting time
Prepare defence & presentation
Thesis Defence

IV

Research question

Research question and objectives

This research focuses on the neural representation of time-optimal quadrotor flight. This research focuses only on the guidance and control problem of time-optimal flight. In order to fully deploy autonomous drones, one would have to also address the state estimation problem. As estimating states is usually treated as a whole separate problem, using for instance visual odometry to determine the drone's position and attitude, it is not treated in this research, hence it is assumed that accurate state estimates are available (using the motion capture system Opti-Track).

Previous work [3] has already shown that G&CNETs trained using supervised learning can successfully guide and control drones but do not reach time optimality yet. Hence, this research will focus on bridging the reality gap in order to fly faster while retaining the robustness of existing G&CNETs. This research comprises of tests in simulation and real flights at the Cyberzoo (TU Delft). Given the limited computational power onboard of quadrotors, the G&CNET resulting from this work needs to be light enough to be run onboard. That being said, the research question and objective are formulated below.

4.5. Research Question

The main research question of this thesis is:

> "Can current G&CNETs perform more time-optimally without losing robustness and still meet the computational requirement to be run onboard of a quadrotor?".

4.6. Research Objective

The main research objective of this thesis is:

> "To improve current G&CNETs in terms of their time-optimality by reducing the reality gap introduced by model inaccuracies and the optimal control formulation.".

To bridge the reality gap caused by modelling inaccuracies the idea presented by Robin Ferede will be extended. Robin uses domain randomization to train the G&CNET on a range of external moment disturbances. This research will include corrections for the drag forces and thrust, as well as corrections in actuator delay.

To bridge the gap caused by the optimal control formulation, optimal trajectories passing through two consecutive waypoints will be considered. In addition, flight data will be analysed to come up with a novel cost function which facilitates the OCP.

The relevance of this work is apparent in the numerous applications of quadrotors, some of which depend on fast flying capabilities to be successful. Search and rescue missions, delivery service, cinematography and drone racing are only a few of these applications. In addition, this research is relevant as there aren't many methods to systematically improve G&CNETs. Performing extensive system identification will improve the performance of a G&CNET but transfers poorly when a different quadrotor needs to be used and is not robust to changes in aerodynamic properties. Understanding better what drives the performance of machine learning for control applications has repercussions on a myriad of robotic applications.

Bibliography

[1] Foehn, P., Romero, A., and Scaramuzza, D., "Time-optimal planning for quadrotor waypoint flight," *Science Robotics*, Vol. 6, No. 56, 2021. doi:10.1126/scirobotics.abh1221, URL https://doi.org/10.1126%2Fscirobotics.abh1221.

[2] Salzmann, T., Kaufmann, E., Pavone, M., Scaramuzza, D., and Ryll, M., "Neural-MPC: Deep Learning Model Predictive Control for Quadrotors and Agile Robotic Platforms," , 2022. doi:10.48550/ARXIV.2203.07747, URL https://arxiv.org/abs/2203.07747.

[3] Li, S., Ozturk, E., Wagter, C. D., de Croon, G. C. H. E., and Izzo, D., "Aggressive Online Control of a Quadrotor via Deep Network Representations of Optimality Principles," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020. doi:10.1109/icra40945.2020.9197443, URL https://doi.org/10.1109%2Ficra40945.2020.9197443.

[4] Kaufmann, E., Loquercio, A., Ranftl, R., Mueller, M., Koltun, V., and Scaramuzza, D., "Deep Drone Acrobatics," 2020. doi:10.15607/RSS.2020.XVI.040.

[5] Verbeke, J., and Schutter, J. D., "Experimental maneuverability and agility quantification for rotary unmanned aerial vehicle," *International Journal of Micro Air Vehicles*, Vol. 10, No. 1, 2018, pp. 3–11. doi:10.1177/1756829317736204, URL https://doi.org/10.1177/1756829317736204.

[6] Sargent, R., "Optimal control," *Journal of Computational and Applied Mathematics*, Vol. 124, No. 1, 2000, pp. 361–371. doi:https://doi.org/10.1016/S0377-0427(00)00418-0, URL https://www.sciencedirect.com/science/article/pii/S0377042700004180, numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[7] Pesch, H. J., "The Princess and Infinite-Dimensional Optimization," *Documenta Mathematica*, 2012.

[8] Bellman, R., *Dynamic Programming*, Dover Publications, 1957.

[9] Todorov, E., "Optimal control theory," *Bayesian brain: probabilistic approaches to neural coding*, 2006, pp. 268–298.

[10] Pontryagin, L., *Mathematical Theory of Optimal Processes*, CRC Press, 1987.

[11] Rao, A. V., "A survey of numerical methods for optimal control," *Advances in the Astronautical Sciences*, Vol. 135, No. 1, 2009, pp. 497–528.

[12] Bellman, R., and Dreyfus, S., "Functional Approximations and Dynamic Programming," *Mathematical Tables and Other Aids to Computation*, Vol. 13, No. 68, 1959, pp. 247–251. URL http://www.jstor.org/stable/2002797.

[13] Bliss, G., *Lectures on the calculus of variations*, University of Chicago Press, 1961.

[14] Bryson, J. A. E., and Ho, Y.-C., *Applied Optimal Control: Optimization, Estimation, and Control*, Taylor & Francis Group, 1975.

[15] Rösmann, C., Makarow, A., and Bertram, T., "Time-optimal control with direct collocation and variable discretization," , 2020. doi:10.48550/ARXIV.2005.12136, URL https://arxiv.org/abs/2005.12136.

[16] Kelly, M., "An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation," *SIAM Review*, Vol. 59, No. 4, 2017, pp. 849–904. doi:10.1137/16M1062569, URL https://doi.org/10.1137/16M1062569.

[17] Foehn, P., and Scaramuzza, D., "Onboard State Dependent LQR for Agile Quadrotors," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2018. doi:10.1109/icra.2018.8460885, URL https://doi.org/10.1109%2Ficra.2018.8460885.

[18] Camacho, E., and Bordons, C., *Model Predictive Control*, Vol. 13, 2004. doi:10.1007/978-0-85729-398-5.

[19] Kaiser, E., Kutz, J. N., and Brunton, S. L., "Sparse identification of nonlinear dynamics for model predictive control in the low-data limit," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, Vol. 474, No. 2219, 2018, p. 20180335. doi:10.1098/rspa.2018.0335, URL https://royalsocietypublishing.org/doi/abs/10.1098/rspa.2018.0335.

[20] Torrente, G., Kaufmann, E., Foehn, P., and Scaramuzza, D., "Data-Driven MPC for Quadrotors," *CoRR*, Vol. abs/2102.05773, 2021. URL https://arxiv.org/abs/2102.05773.

[21] Narendra, K., "Neural networks for control theory and practice," *Proceedings of the IEEE*, Vol. 84, No. 10, 1996, pp. 1385–1406. doi:10.1109/5.537106.

[22] Qin, T., *Machine Learning Basics*, Springer Singapore, Singapore, 2020, pp. 11–23. doi:10.1007/978-981-15-8884-6_2, URL https://doi.org/10.1007/978-981-15-8884-6_2.

[23] Azar, A. T., Koubaa, A., Ali Mohamed, N., Ibrahim, H. A., Ibrahim, Z. F., Kazim, M., Ammar, A., Benjdira, B., Khamis, A. M., Hameed, I. A., and Casalino, G., "Drone Deep Reinforcement Learning: A Review," *Electronics*, Vol. 10, No. 9, 2021. doi:10.3390/electronics10090999, URL https://www.mdpi.com/2079-9292/10/9/999.

[24] Foehn, P., Romero, A., and Scaramuzza, D., "Time-Optimal Planning for Quadrotor Waypoint Flight," *CoRR*, Vol. abs/2108.04537, 2021. URL https://arxiv.org/abs/2108.04537.

[25] Romero, A., Sun, S., Foehn, P., and Scaramuzza, D., "Model Predictive Contouring Control for Time-Optimal Quadrotor Flight," , 2021. doi:10.48550/ARXIV.2108.13205, URL https://arxiv.org/abs/2108.13205.

[26] Foehn, P., Brescianini, D., Kaufmann, E., Cieslewski, T., Gehrig, M., Muglikar, M., and Scaramuzza, D., "AlphaPilot: Autonomous Drone Racing," , 2020. doi:10.48550/ARXIV.2005.12813, URL https://arxiv.org/abs/2005.12813.

[27] Mellinger, D., and Kumar, V. R., "Minimum snap trajectory generation and control for quadrotors," *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 2520–2525.

[28] Mollick, E., "Establishing Moore's Law," *IEEE Annals of the History of Computing*, Vol. 28, No. 3, 2006, pp. 62–75. doi:10.1109/MAHC.2006.45.

[29] Romero, A., Penicka, R., and Scaramuzza, D., "Time-Optimal Online Replanning for Agile Quadrotor Flight," , 2022. doi:10.48550/ARXIV.2203.09839, URL https://arxiv.org/abs/2203.09839.

[30] Bardi, M., Dolcetta, I. C., et al., *Optimal control and viscosity solutions of Hamilton-Jacobi-Bellman equations*, Vol. 12, Springer, 1997.

[31] Sánchez-Sánchez, C., and Izzo, D., "Real-time optimal control via deep neural networks: study on landing problems," *Journal of Guidance, Control, and Dynamics*, Vol. 41, No. 5, 2018, pp. 1122–1135.

[32] Izzo, D., and Öztürk, E., "Real-Time Guidance for Low-Thrust Transfers Using Deep Neural Networks," *Journal of Guidance, Control, and Dynamics*, Vol. 44, No. 2, 2021, pp. 315–327. doi:10.2514/1.G005254.

[33] Tailor, D., and Izzo, D., "Learning the optimal state-feedback via supervised imitation learning," *Astrodynamics*, Vol. 3, No. 4, 2019, pp. 361–374.

[34] Izzo, D., and Origer, S., “Neural representation of a time optimal, constant acceleration rendezvous,” , 2022. doi:10.48550/ARXIV.2203.15490, URL https://arxiv.org/abs/2203.15490.

[35] Song, Y., Steinweg, M., Kaufmann, E., and Scaramuzza, D., “Autonomous Drone Racing with Deep Reinforcement Learning,” , 2021. doi:10.48550/ARXIV.2103.08624, URL https://arxiv.org/abs/2103.08624.

[36] Nagami, K., and Schwager, M., “HJB-RL: Initializing Reinforcement Learning with Optimal Control Policies Applied to Autonomous Drone Racing,” *Robotics: Science and Systems XVII*, 2021.

[37] Carmona, M. A., Munishkin, A. A., Boivin, M., and Milutinović, D., “Stochastic Optimal Approach to the Steering of an Autonomous Vehicle through a Sequence of Roadways,” *2019 American Control Conference (ACC)*, 2019, pp. 3279–3284. doi:10.23919/ACC.2019.8814762.

[38] Loquercio, A., Kaufmann, E., Ranftl, R., Dosovitskiy, A., Koltun, V., and Scaramuzza, D., “Deep Drone Racing: From Simulation to Reality With Domain Randomization,” *IEEE Transactions on Robotics*, Vol. PP, 2019, pp. 1–14. doi:10.1109/TRO.2019.2942989.

[39] Scheper, K. Y. W., and de Croon, G. C. H. E., “Abstraction, Sensory-Motor Coordination, and the Reality Gap in Evolutionary Robotics,” *Artificial Life*, Vol. 23, No. 2, 2017, pp. 124–141. doi:10.1162/ARTL_a_00227, URL https://doi.org/10.1162/ARTL_a_00227.

Part III

Addition results

Addition results

Introduction

This Chapter describes additional results which might come in handy to anyone who wants to continue this project. The following topics are covered: tips are provided on how to solve complex trajectories, trajectories with a high number of nodes and improving the quality of the training dataset when using AMPL. Flight data of a real flight is provided as well as a discussion of which parts of the dynamic model fail when moving towards time-optimal flight. Findings on how to improve the guidance & control network training procedure are described. A comparison between the energy-optimal and the time-optimal objectives is provided. Finally, an example of a real flight where the waypoints are set to different altitudes is given.

Solving optimal trajectories in AMPL

In order to generate the training datasets for the Guidance & Control Networks (G&CNETs), thousands of optimal trajectories are generated using the modelling language AMPL. This section explains how AMPL can be used to solve complex trajectories, increase the amount of nodes for a given trajectory and gives possible ways to improve the quality of the training data.

Solving complex trajectories

Solving complete tracks is interesting for two reasons: it allows to see how close the quadcopter is flying to global optimal trajectory, even when it has only learned to fly to a specific waypoint or to fly a turn. In addition, for time-optimal flight, it gives an indication for what the optimal time of flight is for an entire track. In the case of the Parrot Bebop 1, the minimum time of flight for the 4×3m track is 2.57s when starting from hover and using a waypoint distance constraint of 1cm.

AMPL can easily solve short trajectories and relatively simple optimal control problems using the 6-degrees-of-freedom quadcopter model. However, in the current setup it becomes difficult for the solver to converge as soon as the tracks become more complex if no good initial guess is provided. We experimented with three approaches, all of which use continuation (i.e. using the solution of a similar optimal control problem as an initial guess) to aid the solver. Consider solving the 4×3m track used in this work in one go by setting four intermediate waypoint constraints as described in Part.I. If the waypoint constraint (distance from the intermediate waypoints) is too small and no good initial guess is provided, then AMPL does not converge. A way to mitigate this is by using constraint relaxation. By first setting the waypoint distance constraint to 1m and then using continuation, one can incrementally enforce the solution to pass closer to the waypoint (up until 1cm, see Fig.1.1). Continuation can also be used by combining different solutions together. Fig.1.2 shows how the 4×3m track is solved by first individually solving each leg of the track (each portion between two consecutive waypoints). The solutions of all four legs are then "glued" together and used as an initial guess to solve the complete track. Finally, it is also possible to solve each turn first and then "glue" the solutions together, see Fig.1.3. This last method also works well for even more complex tracks. We managed to solve a track containing seven waypoints with left and right turns by first solving each turn individually. Without continuation such tracks cannot be solved directly by AMPL.

The final solutions for the entire track in Fig.1.1, Fig.1.2 and Fig.1.3 are actually all slightly different. This is likely due to a variety of reasons:

- We are using a direct method, hence the solutions are approximations of the true optimal trajectory.
- The solutions of the solver have numerical noise due to integration errors between the nodes.
- We might be stuck in a local minimum.
- The way in which ones uses continuation strongly influences the quality of the final solution.

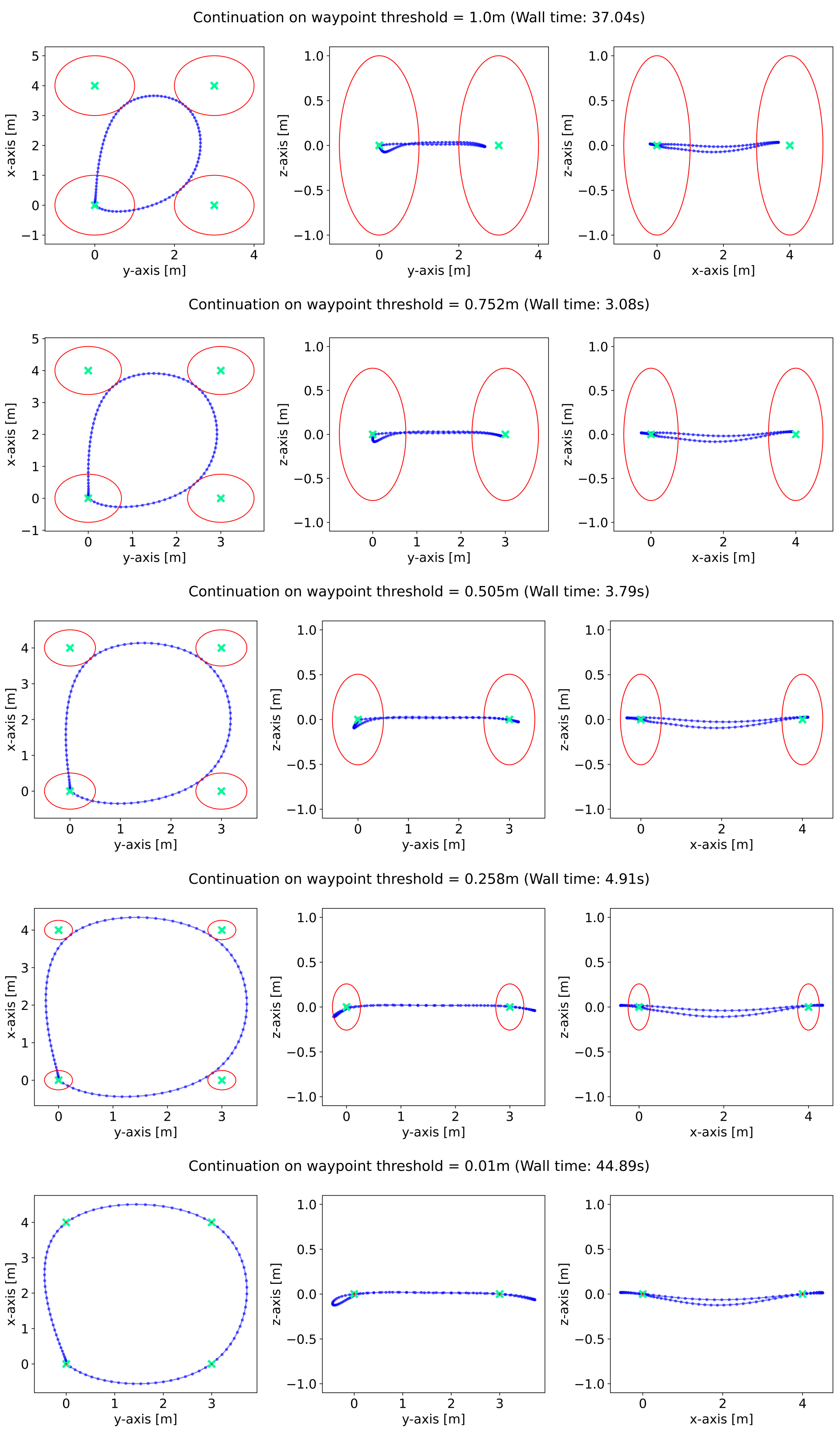


Figure 1.1: From top to bottom: waypoint distance constraint relaxation to solve the entire track, then using continuation, the subsequent optimal control problems can be solved. The green crosses and red circles indicate the waypoint positions and distance constraint, respectively.

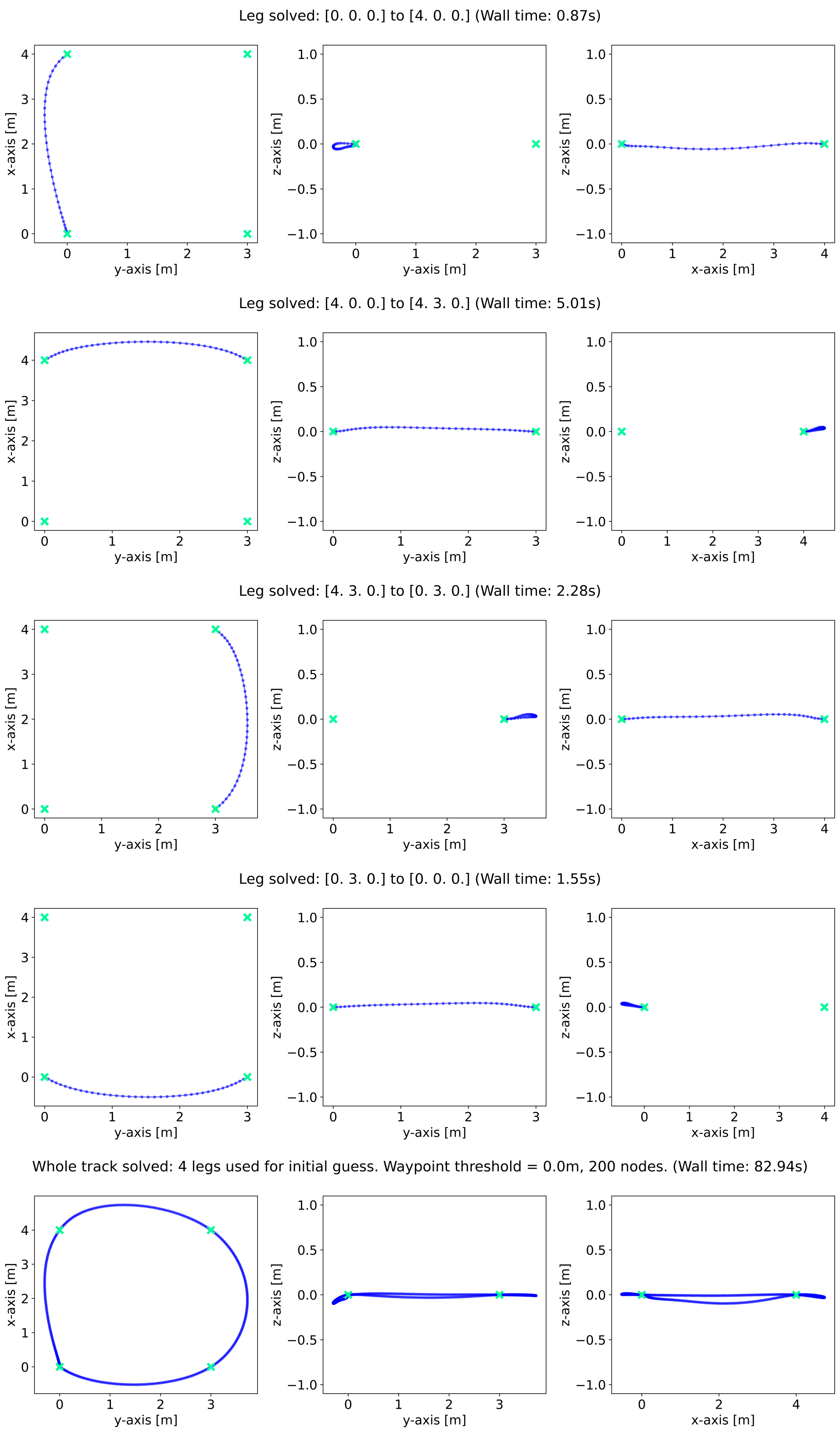


Figure 1.2: From top to bottom: solving the entire 4×3m track by first solving each leg (the trajectories between each waypoints) and then glueing the solutions together to provide the solver with a good initial guess. The green crosses indicate the waypoint positions.

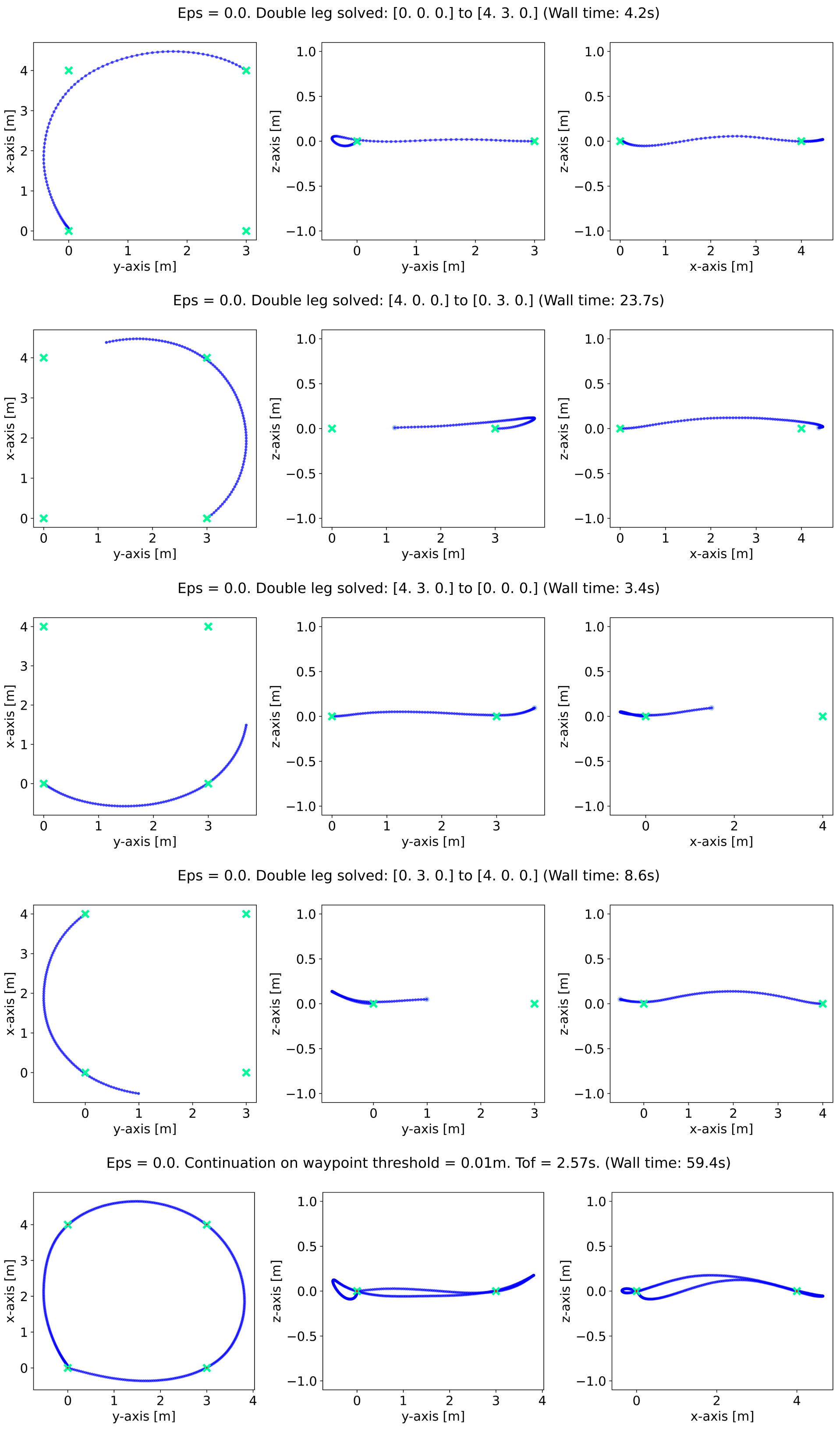


Figure 1.3: From top to bottom: solving the entire 4×3m track by first solving each turn and then glueing the solutions together to provide the solver with a good initial guess. The green crosses indicate the waypoint positions.

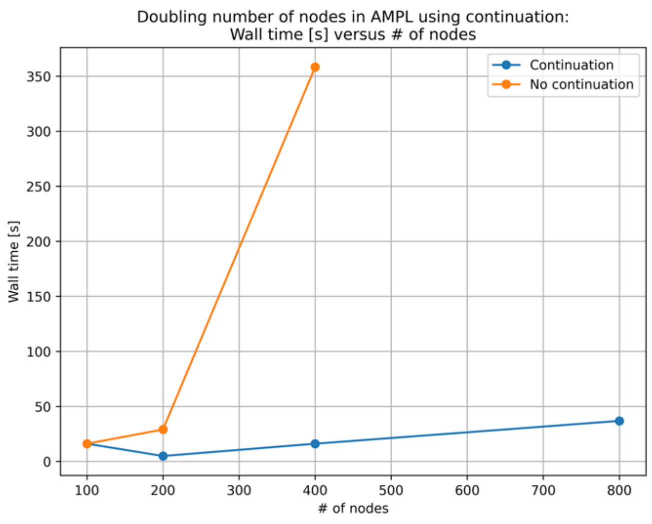


Figure 1.4: Wall time of solver versus number of nodes of optimal control problem.

Node doubling technique

The amount of nodes over which the optimal control problem is discretized has also shown to strongly affect convergence and the solution itself. As described in Part.I, increasing the number of nodes might be desirable to alleviate some of the numerical noise present in the solution. This can be done by first solving the problem for a low number of nodes, then interpolating the optimal solution with quadratic splines and projecting this interpolant on a new grid of nodes. Using continuation, it is for instance possible to solve optimal trajectories with 1000 nodes, which is not possible if no good initial guess is provided. In addition, a major disadvantage of machine learning technique is that generating training data can take a lot of time. By using continuation one can drastically decrease the wall time needed for the solver to converge. Consider Fig.1.4, where we first solve a given optimal control problem with 100 nodes. Using continuation it is possible to solve this same trajectory with 800 nodes in 40s. Without continuation AMPL does not even converge for 800 nodes and for 400 nodes it takes almost 6min to find a solution.

Possible ways to improve training dataset quality

Since the network tries to approximate all training trajectories as best of possible, it is conceivable that sub-optimal trajectories have a detrimental effect on the accuracy of G&CNETs. One way to decrease the likelihood of local minima in the training dataset is to solve a given trajectory multiple times with AMPL, each time giving a different random initial guess. In case AMPL converges to the same solution most of the time, one can be more confident that this solution is close to the global optimum. Now that we have one "good" solution, we can use continuation to solve optimal control problems with neighbouring initial conditions. This way the quality of each solution is increased and the resulting training dataset is more likely to be coherent. In some rare cases, multiple optimal paths exist, for instance flying around an object from the left or the right side. Solving such an optimal control problem multiple times would prevent injecting ambiguous cases to the training dataset since the solutions would not converge often enough to the same solution.

In the current setup the initial conditions for the optimal trajectories are uniformly sampled within certain bounds. One way to improve the learning process and the performance of the resulting G&CNET could be to generate training datasets which are tailored to the track. This work has not explored whether or not a uniform distribution of initial conditions is necessarily required. One way one can tailor the training dataset is by testing an existing G&CNET in simulation or on the real quadcopter. Every point in state-space of the flight data is a valid initial condition for the training dataset. Care should be taken as AMPL sometimes converges to weird solutions that are clearly sub-optimal, however these are considered successful solutions by the solver. By solving similar trajectories and comparing these one can spot and filter out these sub-optimal trajectories from the training dataset which will likely result in a lower loss during training.

Time-optimal flight with G&CNETs: shortcomings

It is possible to fly the Parrot Bebop 1 with a G&CNET trained with $\epsilon = 0.35$ using the following cost function:

$$J(\mathbf{u}, T) = (1 - \epsilon)T + \epsilon \int_0^T ||\mathbf{u}(t)||^2 dt$$

where $\mathbf{u}$ are the controls and T is the time-of-flight. However, in the current setup flights with $\epsilon < 0.5$ do not fly the 4×3m track consistently on the same path. Part.I explains why some of these instabilities are due to the loss during training. In this section we eloborate a bit more on the reality gap which is also a strong contributor to unstable flight. This gap is mainly caused by delays in the hardware, state estimation errors with the Optitrack system and modelling errors. Fig.1.5 shows the velocity map of a G&CNET with $\epsilon = 0.4$ (hence only slightly more time-optimal than the one used in Part.I). Clearly the network struggles to keep the quadcopter on the optimal path. The commanded and observed angular velocities of all four rotors during the flight are shown in Fig.1.6. The two back rotors (ω_3 and ω_4) saturate during a considerable portion of the flight. Compared to the flight with $\epsilon = 0.5$ in Part.I there is even less control authority left to correct for errors.

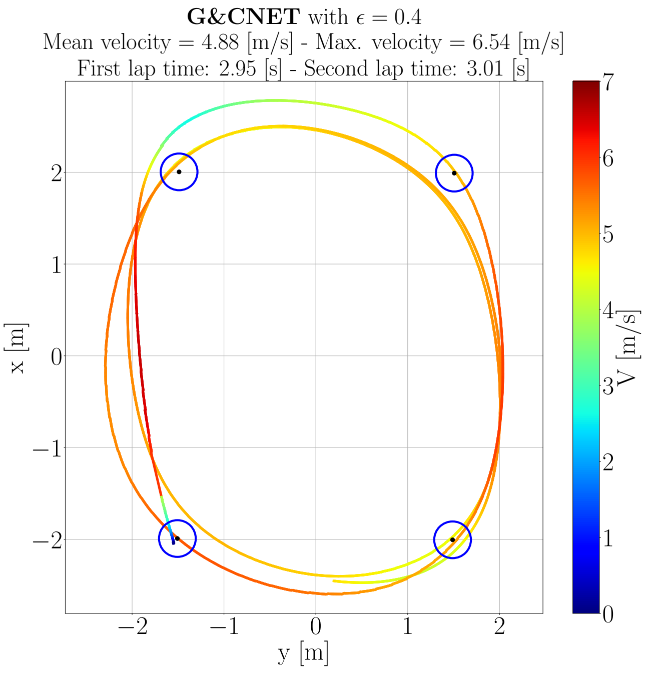


Figure 1.5: Trajectory of a real flight with $\epsilon = 0.5$.

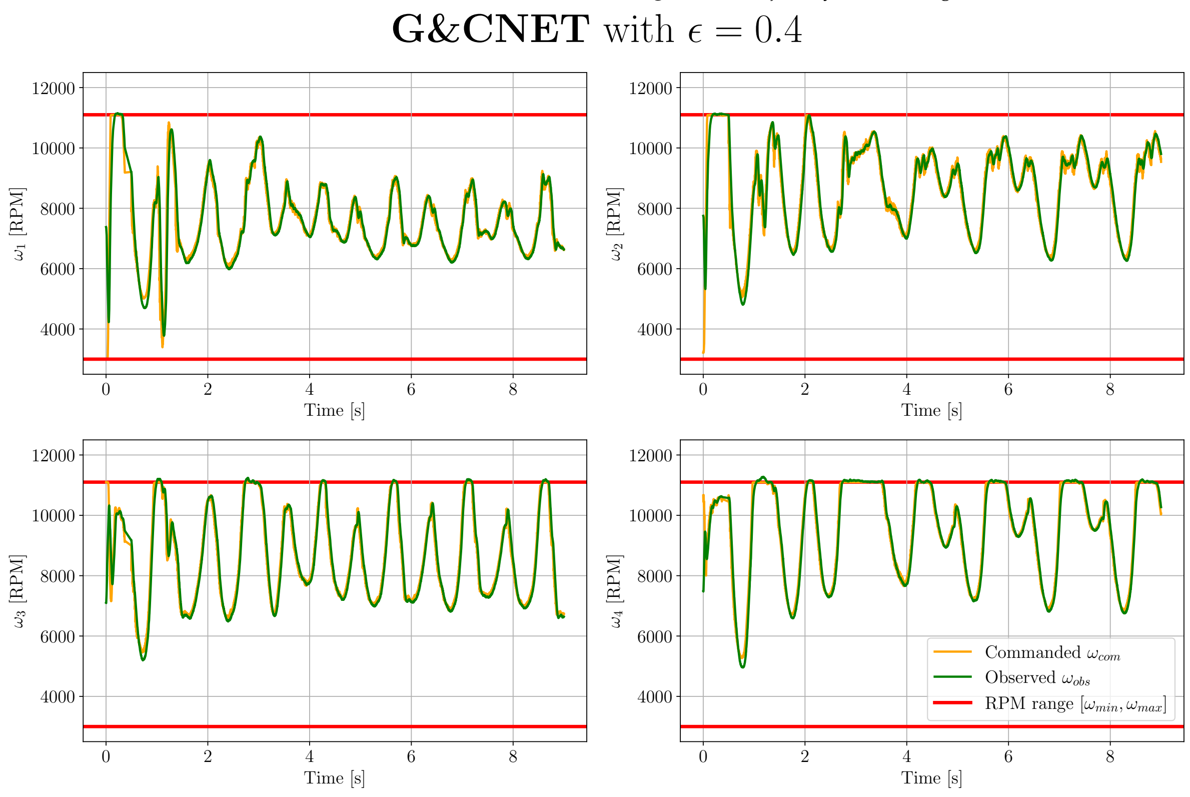


Figure 1.6: Commanded and observed angular velocities of the four rotors.

Fig.1.7 shows the errors between modeled and measured moments which are fed to the adaptive G&CNET, following the method proposed by [3]. While the G&CNET learned to take into account these modeling errors there are a couple of reasons why the current setup might be insufficient to fly stably:

- One of the main assumption of the method proposed in [3] is the there is a constant external moment disturbance acting on the drone, which is not the case in Fig.1.7.
- The error that is fed to the network is delayed since it is low-pass filtered (cut-off frequency of 9Hz). However the quadcopter is doing aggressive maneuvers, leading to rapid changes in the sign of the error (see $t = 1$s in Fig.1.7).
- The magnitudes of the moment errors around the y-axis and z-axis are outside of the bounds which are used to generate the training dataset ($M_{ext,y} \in [-0.04, 0.04]$, Nm and $M_{ext,z} \in [-0.01, 0.01]$, Nm).

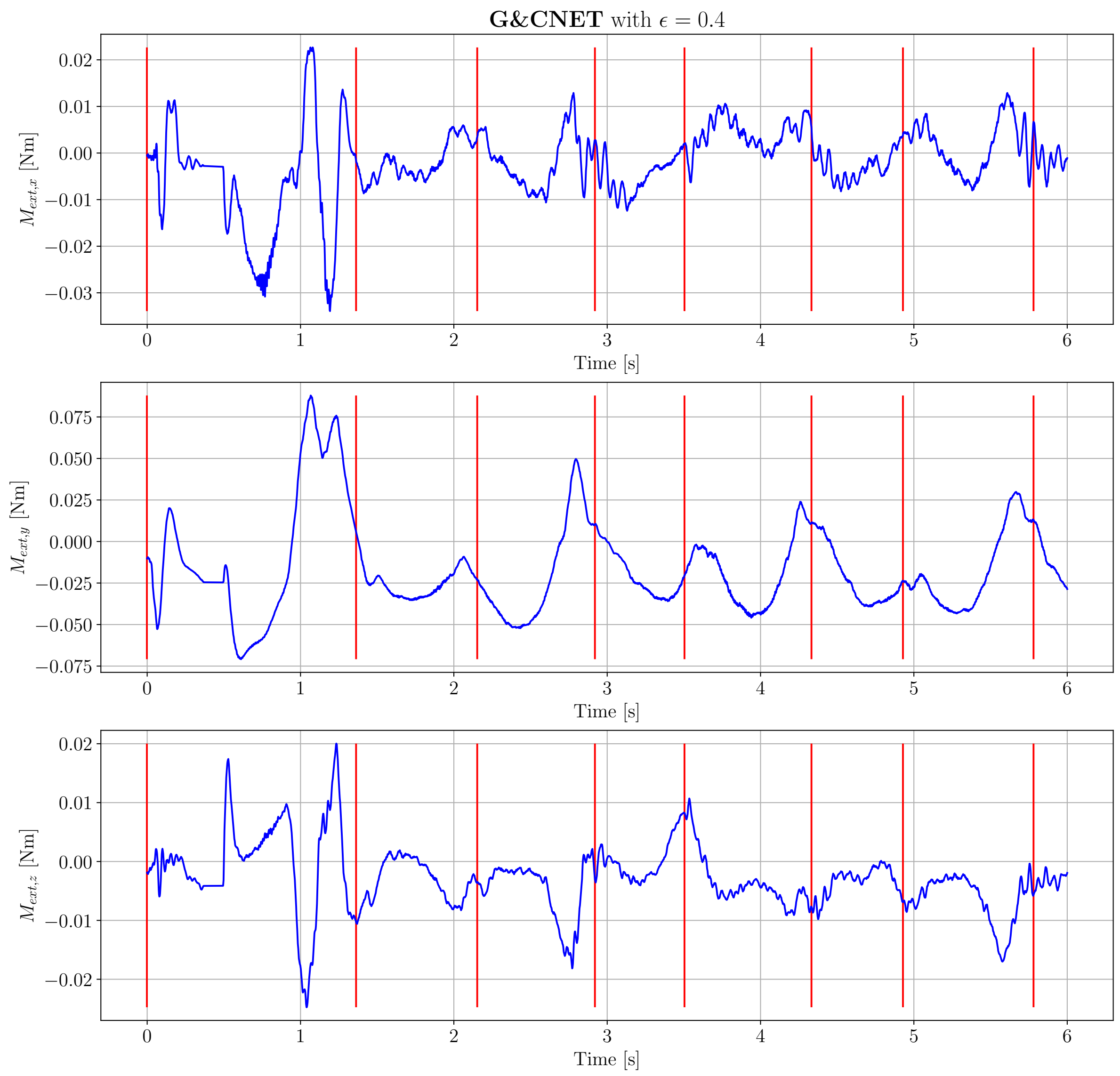


Figure 1.7: Differences between modelled and measured moments along all three axes. These signals are fed to the G&CNET following the adaptive method proposed in [3]. The red vertical lines roughly indicate the times at which the quadcopter passes a waypoint.

The quadcopter model used in this work also does not take into account its downwash, which could be one of the causes for the model errors in acceleration. We noticed that the drone particularly had difficulties maintaining its altitude for lower ϵ. Fig.1.8 shows the measured and modeled acceleration in the z-axis (world frame, see Part.I). It might be beneficial to either perform more in depth system identification in the future or one could try a similar approach as in [3] only this time for thrust and drag, i.e. using domain randomization for the training trajectories by adding external forces disturbances. This model error (difference between the two signals shown in Fig.1.8) can then be computed onboard and fed to the G&CNET.

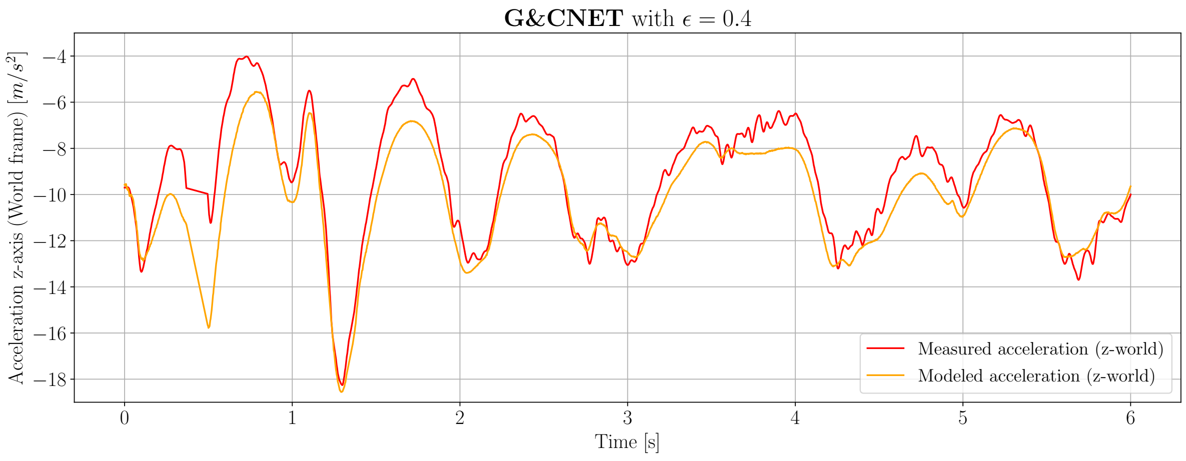


Figure 1.8: Measured and modeled acceleration in the z-axis (world coordinate system, see Part.I).

G&CNET training procedure

The standard network architecture used in this work is shown in Fig.1.9. This section provides the finding when comparing the final loss (mean squared error) of different training setups. The current architecture is reaching its limit in terms of the amount of information it can learn to approximate. Using the same training setup and only adding one hidden layer or doubling the amount of neurons in the hidden layer have both shown to result in a lower loss. The G&CNET also stands to benefit from larger training datasets. We tried dataset with 10000, 40000 and 100000 optimal trajectories, in every case the loss decreased for larger dataset. We also tried using Softplus activation functions instead of ReLU for the hidden layers, as Softplus are advantageous to obtain a continuous representation of the optimal controls. There was no apparent difference in final loss when comparing both activation functions, hence we kept the ReLU as the network can be inferred faster onboard of the quadcopter with this activation function. Future work could try to further improve the loss by training longer, using different batch-sizes and rigorously removing all sub-optimal trajectories in the training dataset.

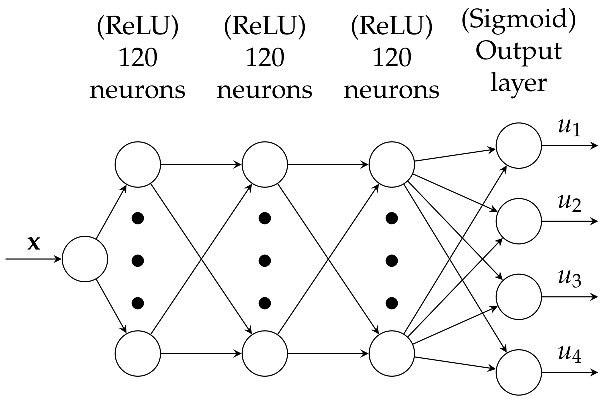


Figure 1.9: Feed forward network architecture.

Comparison between energy- and time-optimal objectives

A rough indication of how much time there is to gain when moving from the energy-optimal control problem to the time-optimal control problem is provided here. Let's consider flying in a straight line 5m forward, starting from hover. The resulting trajectories for a range of $\epsilon \in [0.0, 1.0]$ are shown in Fig.1.11 where the coloring of the trajectories corresponds to the one used in Fig.1.10, which shows the time-of-flight for each of them. The time-optimal trajectory has considerably larger excursions in the z-direction (Fig.1.11) and its time-of-flight is 0.5s lower than for the energy-optimal case. While minimizing energy indirectly minimizes the time spent in the air, it is still considerably slower than the time-optimal objective.

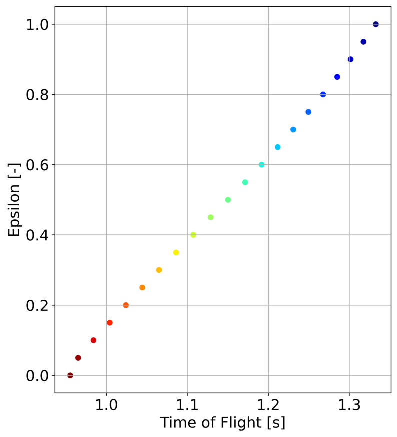


Figure 1.10: Epsilon versus time-of-flight. Corresponding trajectories are shown in Fig.1.11.

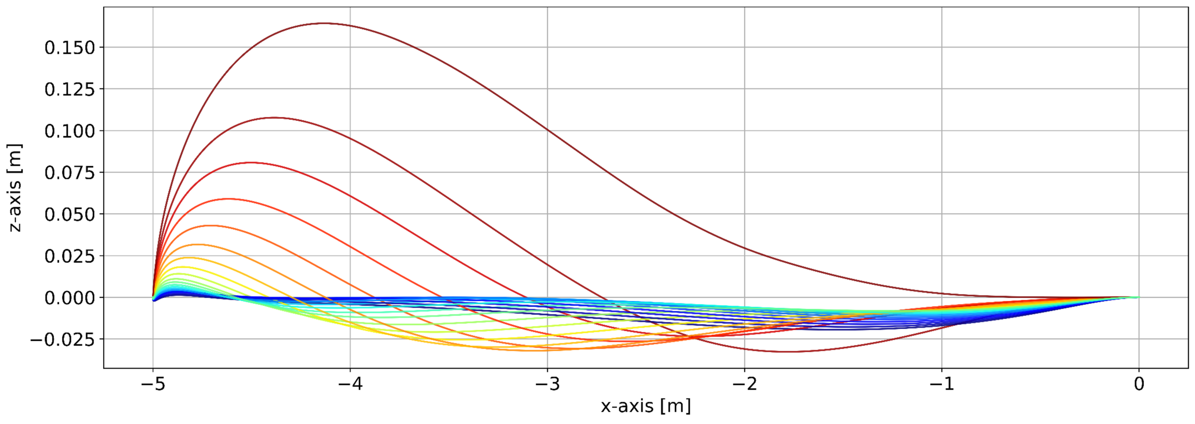


Figure 1.11: Optimal trajectories ranging from energy-optimal (blue) to time-optimal (red).

Flying at different altitudes with a G&CNET

Finally, in this entire work the flights are kept in the XY-plane for simplicity. However, it is definitely possible to fly at different altitudes. This is shown in Fig.1.12 which is a real flight using $\epsilon = 0.4$. The waypoints are positioned such that the quadcopter incrementally spirals up and then down by 1m in total. The reasons why larger altitude differences where not yet tested are because the initial altitudes for the training dataset are sampled within a small range ($z_0 \in [-0.5, 0.5]$, m) and to avoid losing track of the quadcopter in the Cyberzoo (which can happen close to the ceiling).

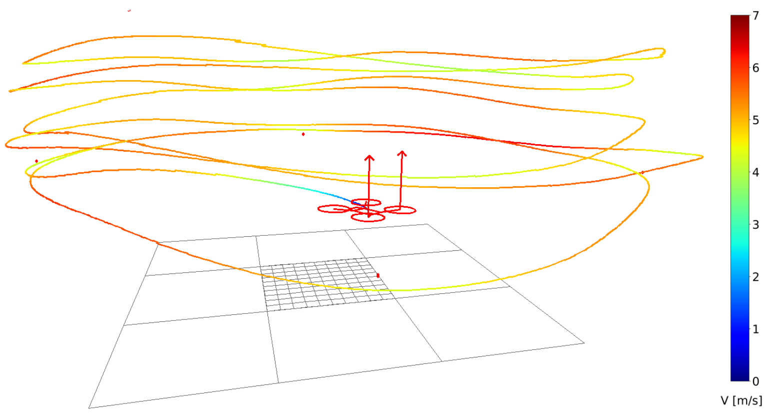


Figure 1.12: Real flight with $\epsilon = 0.4$. The quadcopter spirals up and down by 1m.

References

[1] Leonard Bauersfeld and Davide Scaramuzza. *Range, Endurance, and Optimal Speed Estimates for Multicopters*. 2021. DOI: 10.48550/ARXIV.2109.04741. URL: https://arxiv.org/abs/2109.04741.

[2] C. De Wagter et al. "Learning fast in autonomous drone racing". English. In: *Nature Machine Intelligence* 3.10 (2021). Copyright: Copyright 2021 Elsevier B.V., All rights reserved., p. 923. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00405-z.

[3] Robin Ferede et al. "An Adaptive Control Strategy for Neural Network based Optimal Quadcopter Controllers". MA thesis. TU Delft Aerospace Engineering, 2022. URL: http://resolver.tudelft.nl/uuid:b43a9703-082c-47c7-a56e-d50794ee8c1c.

[4] Philipp Foehn, Angel Romero, and Davide Scaramuzza. "Time-optimal planning for quadrotor waypoint flight". In: *Science Robotics* 6.56 (2021), eabh1221. DOI: 10.1126/scirobotics.abh1221. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abh1221. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abh1221.

[5] Dario Izzo and Sebastien Origer. "Neural representation of a time optimal, constant acceleration rendezvous". In: *Acta Astronautica* (2022). ISSN: 0094-5765. DOI: https://doi.org/10.1016/j.actaastro.2022.08.045. URL: https://www.sciencedirect.com/science/article/pii/S0094576522004581.

[6] Dario Izzo and Ekin Öztürk. "Real-Time Guidance for Low-Thrust Transfers Using Deep Neural Networks". In: *Journal of Guidance, Control, and Dynamics* 44.2 (2021), pp. 315–327. DOI: 10.2514/1.G005254.

[7] Elia Kaufmann et al. "Deep Drone Acrobatics". In: *RSS: Robotics, Science, and Systems* (2020).

[8] Shuo Li et al. "Aggressive Online Control of a Quadrotor via Deep Network Representations of Optimality Principles". In: *CoRR* abs/1912.07067 (2019). arXiv: 1912.07067. URL: http://arxiv.org/abs/1912.07067.

[9] Shuo Li et al. "Autonomous drone race: A computationally efficient vision-based navigation and control strategy". In: *Robotics and Autonomous Systems* 133 (2020), p. 103621. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2020.103621. URL: https://www.sciencedirect.com/science/article/pii/S0921889020304619.

[10] Giuseppe Loianno et al. "Estimation, Control, and Planning for Aggressive Flight With a Small Quadrotor With a Single Camera and IMU". In: *IEEE Robotics and Automation Letters* 2.2 (2017), pp. 404–411. DOI: 10.1109/LRA.2016.2633290.

[11] Daniel Mellinger and Vijay Kumar. "Minimum snap trajectory generation and control for quadrotors". In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 2520–2525. DOI: 10.1109/ICRA.2011.5980409.

[12] Daniel Mellinger, Nathan Michael, and Vijay Kumar. "Trajectory generation and control for precise aggressive maneuvers with quadrotors". In: *The International Journal of Robotics Research* 31.5 (2012), pp. 664–674. DOI: 10.1177/0278364911434236. eprint: https://doi.org/10.1177/0278364911434236. URL: https://doi.org/10.1177/0278364911434236.

[13] Kartik Mohta et al. "Fast, Autonomous Flight in GPS-Denied and Cluttered Environments". In: *Journal of Field Robotics* 35 (Dec. 2017). DOI: 10.1002/rob.21774.

[14] Angel Romero et al. *Model Predictive Contouring Control for Time-Optimal Quadrotor Flight*. 2021. DOI: 10.48550/ARXIV.2108.13205. URL: https://arxiv.org/abs/2108.13205.

[15] Tim Salzmann et al. *Neural-MPC: Deep Learning Model Predictive Control for Quadrotors and Agile Robotic Platforms*. 2022. DOI: 10.48550/ARXIV.2203.07747. URL: https://arxiv.org/abs/2203.07747.

[16] Carlos Sánchez-Sánchez and Dario Izzo. “Real-Time Optimal Control via Deep Neural Networks: Study on Landing Problems”. In: *Journal of Guidance, Control, and Dynamics* 41 (Oct. 2016). doi: `10.2514/1.G002357`.