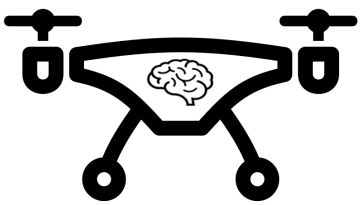
Low level quadcopter control using Reinforcement Learning

Developing a self-learning drone

Tim Koning 4095286



TU Delft

Low level quadcopter control using Reinforcement Learning

Developing a self-learning drone

by

Tim Koning 4095286

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday April 14th, 2020

Student number:	4095286	
Project duration:	feb, 2019 – april, 2020	
Thesis committee:	Dr. W. Pan,	TU Delft, daily supervisor
	Prof. dr. ir. M. Wisse,	TU Delft, c hair
	Dr. ing. J. Kobe,	TU Delft

TU Delft

Abstract

Background

Reinforcement Learning (RL) is a learning paradigm where an agent learns a task by *trial and error*. The agent needs to explore its environment and by simultaneously receiving rewards it learns what is appropriate behaviour. Even though it has roots in machine learning, RL is essentially different from other machine learning methods. In contrast to others, RL agent has to generate its own data to learn from.

Thesis goal

In this thesis, we aim to train an RL agent to fly a quadcopter to track any target position (way-point) in three dimensional space. Where conventional control strategies for quadcopters involve a separate attitude and position controller and most RL solutions focus on one of the two controllers, our goal is to design a low level RL controller capable of computing motor commands directly from sensor input, therefore replacing both attitude and position controller with one RL policy[1].

Method

The agent learns in simulation, for which a simulation model is needed. This simulation model consists of the quadcopter dynamics model using the parameters resulting from a proposed two-step parameter estimation. The first step of the two-step parameter estimation consists of measuring the measurable parameters of the quadcopter and estimation of the inertia matrix by a summation of multiple point masses and a central cylindrical mass. The second step is recreating a physical model including a controller in Simulink and using the Simulink parameter estimation toolbox to fine-tune the estimated parameters to fit real flight data.
The policies we develop utilize the algorithm 'Twin Delayed Deep Deterministic Policy Gradient' (TD3) for learning. TD3 is a variant of the Deep Deterministic Policy Gradient (DDPG) algorithm but incorporates three modifications, making it advantageous with respect to DDPG. The modifications are target policy smoothing, clipped double Q-Learning and delayed policy updates. TD3 is capable of handling large state spaces and continuous actions. We develop policies for two main tasks: Attitude [2] control and position control. Both tasks make use of a dense reward structure.

Results

The policy for attitude control trained for 3500 episodes [3], around 6.1e5 time steps. The learned policy is able to stabilize the attitude of the quadcopter (in simulation) with a success rate of 94 % and a rise time of 1.58 s. For position control, two policies are generated with a different dense reward type:

- *type 1* policy is trained with a negative reward on both attitude error as well as position error.
- *type 2* policy is trained with only a negative reward on the position error.

Type 1 produces a working policy after a training period of 7500 (around $3e6$ time steps) episodes with an average reward over 100 episodes of 7580. The resulting policy has high fluctuations in motor commands and therefore oscillating attitude and position values. In none of the evaluation trajectories a steady state value is reached. The type 2 produces a working policy after a shorter training time of 1200 episodes, $1.1e6$ time steps. For all trajectories, this policy achieves steady state for almost each way-point. Steady state error has a maximum value of 0.1 m. An advantage of just using a negative reward on the position error is the freedom of the agent to solve the problem. We only tell the agent *what* goal to reach, not *how* to reach it. Therefore, policy *type 2* is favoured over the *type 1* policy.

Conclusion

This thesis proves that TD3 can be used for low-level quadcopter control, replacing both inner and outer loops of the quadcopter control. Using the dense reward function and applying negative reward on position control only results in a stable policy that can track way-points all throughout the 3D space. Future work requires the two-step parameter estimation to be tested on a real life quadcopter, as well as enrolling the policy onto a real life quadcopter.

[1] the decision-maker of the agent
[2] orientation
[3] simulation loop with a terminal state

Contents

List of Figures

List of Tables

Symbols

$\mathcal{A}$	action space
$\mathcal{S}$	state space
$\mathcal{T}$	transition function
r	reward
γ	discount factor
π	policy
τ	trajectory
G_t	return
$V(s)$	Value function
$Q(s,a)$	Action-Value function
α	rlearning rate
$\nabla_\theta J(\theta)$	Gradient of the cost function
μ	actor network
R	rotation matrix
ϕ	roll
θ	pitch
ψ	yaw
u	linear speed in body frame along x-axis
v	linear speed in body frame along y-axis
w	linear speed in body frame along z-axis
p	angular speed in body frame around x-axis
r	angular speed in body frame around z-axis
ω_b	vector of all three angular speeds in body frame
F	forces
M	moments
I	inertia
m	mass
g	gravitational constant
τ_i	torque around *i* axis
Ω	rotor rotational speed
l	distance motor to body axis
c_t	thrust coëfficient
c_τ	torque coëfficient
K_p	proportional constant
K_d	derivative constant
$\mathcal{N}$	noise

1 Introduction

Reinforcement Learning (RL) is an agent-oriented learning paradigm which sprouted from Artificial Intelligence (AI) [11] [12]. In RL a subject (agent) learns by interacting with the environment and receiving rewards for good behaviour or penalties for bad behaviour [30]. RL is used to solve challenging tasks, including learning intelligent behaviour in complex dynamic environments. RL has proven itself to be able to handle difficult tasks: In 2015, Google DeepMinds AlphaGo used RL to beat 18-time world champion Lee Sedol in a five round match of 'GO' by winning four out of five games [33]. Where IBM's computer 'DeepBlue' used a brute force computation method to beat Kasparov in a game of chess in 1997 [4], this brute force method would not have been sufficient for beating the game of GO, due to the much larger game dimensions. Other games have been solved using RL: DeepMind used RL to achieve superhuman performance on 49 classic Atari games [26]. Because of the ability to handle difficult environments, RL has been deployed in real world settings, for instance in robotics [6] and self driving cars.

Reinforcement Learning is based on an *agent*. The agent is the subject of learning, the one who needs to perform a certain task. In a game of chess it could be thought of to be one of the two players. In the case of a self driving car, the agent could be thought of as the car. The agent needs to explore its environment and by simultaneously receiving rewards it learns what is appropriate behaviour. All key parts of Reinforcement learning come together in the following description:

*In reinforcement learning an **agent** needs to take **actions** in an **environment** to maximize a cumulative **reward**.*

To determine whether the action was good or bad, the environment gives back a reinforcement to the agent. This reinforcements is com known as the *reward*. The agent will try to maximize this reward over the course of learning. It will adapt its behaviour, its decision making, to do so. The decision maker is known as the *policy*.

1.1. Unmanned Aerial Vehicles

An Unmanned Aerial Vehicle (UAV) is an aircraft not carrying a human operator. The UAV is controlled either by remote control or by autonomous control. UAV's can take many shapes, from controlled balloons [19], to the more (in)famous General Atomics MQ-9 Reaper used for military purposes [5]. Thanks to improvement in electronics and control, the consumer market now has access to UAV's in the form of multicopter vehicles, commonly known as 'drones' or 'quadcopters' (in case of four rotors), see figure 1.1. These type of consumer UAV's are relatively small and their employability ranges from amusement racing to visual offshore wind turbine inspection. Open-source platforms such as PX4 and ArduPilot have helped the community to develop the technology and create easy access for everyone to enjoy flying drones.
An increased interest has risen to develop RL controllers for this platform. Developing controllers is not

an easy task and different objectives might need a different type of control. Low level RL control might be suited for this task: If an agent is given a certain objective, the RL agent will develop its own controller for that task. The engineer only needs to specify the objective. The objective can be way-point tracking, package delivery, or flying with low power consumption. The agent aims to find an optimal solution itself. Boundaries can be implemented as forbidden states, which the agent then will stay away from. This can be avoiding dangerous areas, obstacles, or going back to a home-base when batteries are too low, depending on the state. The high adaptability of RL agents make this alternative attractive.

The research conducted last couple of years focus mostly on either *outer* or *inner*-loop control: The outer-loop controller is the high level position controller of the quadcopter, determining where to fly next. The inner-loop controller is the control loop that stabilizes the quadcopter. Developing an outer-loop RL controller is similar to developing a path planning algorithm. The quadcopter is already a self-stabilizing vehicle, that can be controlled. Developing an inner-loop controller is similar to developing a stabilizer for a vehicle. Replacing both inner and outer-loop by an RL agent might give the quadcopter more freedom and more options regarding optimization for different tasks.

Thesis relevance

The tasks this thesis focuses on are also achievable using regular control theory. However, by proving this learning framework is effective in applying RL to a quadcopter platform, doors to more difficult tasks will open. Adding sensors for obstacle detection could help an agent to easily learn avoiding obstacles during way-point tracking. Also highly non-linear acrobatic manoeuvres could be learned using this learning framework. Multi agent coöperation and object manipulation can be experimented with.

Figure 1.1: DJI quadcopter

1.2. research question

The goal of this thesis is to create a Reinforcement Learning agent for a quadcopter, replacing both inner and outer control loops, that is able to lift off and track way-points in three dimensional space. Furthermore a framework for sim-to-real transfer by means of parameter identification is proposed. The research questions therefore are:

1. What are the dominant system dynamics?
2. How are quadcopters controlled?
3. What RL algorithm is best suitable for quadcopter control implementation?
4. How can the sim-to-real gap be bridged?

To answer these questions, multiple RL algorithms are analysed in chapter 2. This chapter explains the concept of Q-learning and the following algorithms DDPG and TD3. TD3 is eventually used to train the agent. Quadcopter dynamics and control is reviewed in chapter 3. From these dynamics, a model will be created for training the RL agent. Chapters 4 and 5 review the build of a quad and the identification parameters for sim-to-real transfer. This sim-to-real transfer method consists of a two-step parameter estimation framework that has been tested in simulation. Chapter 6 implements the RL controller for a quadcopter, using the algorithm as explained in chapter 2 and the model from chapters 3. The results are discussed in chapter 7 and the conclusion follows in chapter 8.

2

Reinforcement Learning

In this chapter a summary of Reinforcement Learning (RL) is given. The Actor-Critic (AC) methods are dissected with mathematical background, in order to understand the final algorithm (TD3) used to train the quadcopter agent. In this overview of methods, first Q-learning will be discussed: one of the most used and simple RL algorithms. After that, Actor-Critic methods such as AC, DDPG and finally TD3 will be discussed. A short summary of deep neural networks will be presented, as this will play a big roll in RL. RL combined with deep neural networks is also commonly referred to as Deep Reinforcement Learning (DRL).

2.1. RL introduction

Reinforcement Learning (RL) is a branch of machine learning and is an agent-oriented learning paradigm that learns by interacting with the environment [30]. The term 'agent' in this sentence refers to your subject in the system: this can be a real life robot, or a virtual spaceship in an Atari game. The environment is everything outside of the agent, with which it interacts. All key parts of Reinforcement learning come together in the following description:

*In reinforcement learning an **agent** needs to take **actions** in an **environment** to maximize a cumulative **reward**.*

Other branches of machine learning, such as supervised learning, rely on labelled data sets to accomplish tasks capable of speech or image recognition. [35]. In contrast with those methods, RL has to generate its own data: The agent needs to explore its environment in order to acquire the data to gather information about its surroundings. The agent needs to explore. During exploration, the agent will receive feedback of its actions in the form of a reward. From the obtained reward, the agent determines what is a good action to take and/or what is a good position in the environment to be in.

The reinforcement learning framework is depicted in figure 2.1. The agent at time-step t is in state s_t and acts on the environment with an action a_t. The environment receives the action and returns the new state s_{t+1} at time-step $t+1$ together with a reward R_{t+1}. From the reward the agent can judge the quality of its move. By exploring the environment, the agent can learn how to interact with the environment and receive higher rewards.

A favored analogy is that of a small child that is learning how to take the first steps: At first the child will try some movements (actions) and fall. This process of trial and error will continue, until the baby makes a successful first step, followed by an overwhelming applause from the parents (this resembles the reward). The baby will then try to remember what it did to take the successful step and try to go even further. The baby learns by collecting data about its environment and learning by *trial and error*. Obviously a grown person walks better than a baby. If the baby would stop learning the minute it reaches the point of 'not falling', it would never learn to walk with the souplesse of an adult. So still new types of actions need to be tried (*explored*) in order to improve the walking skills. This is called the

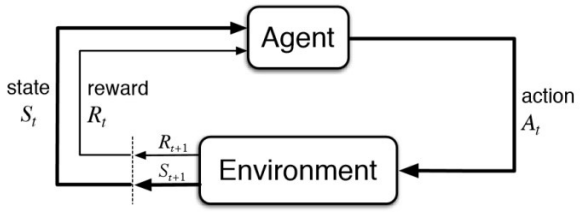


Figure 2.1: Schematic representation of the reinforcement learning framework

exploration vs. exploitation dilemma. This dilemma talks about the decision of an agent to stick to the actions from which it knows will yield a good reward (exploitation), or improving their skill set by trying out new actions and see where it leads them (exploration). Exploitation is also called taking a 'greedy' action. These are the key components of Reinforcement Learning.

2.1.1. Markov Decision Process

The type of agent/environment interaction as described by states, actions, rewards and discounts is called a **Markov Decision Process** or MDP [15]. An MDP is a discrete time stochastic control process and is used in decision making processes where a goal is achieved by learning from interactions. The MDP takes in the current state and takes an action accordingly: no memory of past states are used in this type of problem.

By definition, an MDP exists out of 5 tuples:

- $\mathcal{A}$
- $\mathcal{S}$
- $\mathcal{T}$
- r
- γ

Where $\mathcal{A}$ is the action space containing all possible actions and $\mathcal{S}$ is the state space containing all possible states. $\mathcal{T}$ is the transition function that describes what the next state s_{t+1} will be, when taking action a_t in state s_t. In a probabilistic environment, this transition function describes the probability that taking action a_t in state s_t will lead to the next state s_{t+1} in time $t+1$:

$$\mathcal{T} = Pr(s_{t+1}|s = s_t, a = a_t) \tag{2.1}$$

r is the reward the agent receives after taking an action a_t in state s_t. γ is called the discount factor. This discount factor discounts future rewards and is thus used to determine whether an agent should focus on the short or long term rewards.

The agent needs to decide what action to take, given a certain state and reward. The decision maker of the agent is called the *policy* π. Given a stochastic system, this policy is denoted by $\pi(a|s)$, while the policy given a deterministic system is given by $a = \pi(s)$, where a is an output command of the agent (action) and s is the current state of the agent.

In the new state $t+1$, the policy will create a new action a_{t+1} according to $\pi(a|s)$. Repeating this process will result in a trajectory τ, consisting of state and action pairs.

$$\tau = (s_0, a_0, s_1, a_1, ..., s_n, a_n) \tag{2.2}$$

2.1.2. States and actions

As is also the case in classic control theory, states give you information about the system. In the inverted pendulum problem useful states are the position of the cart and the angle of the pendulum with respect to vertical [7], see figure 2.2. In a more difficult system, for instance a quadcopter, more states can be included to provide a more useful summary of states. Sensors can be added to the system to

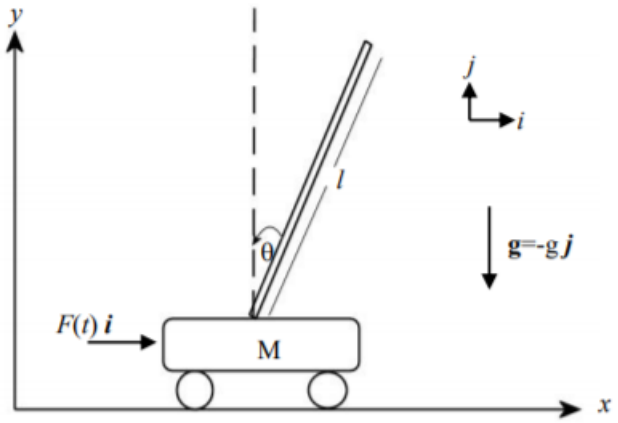


Figure 2.2: the cartpole problem described with states x and θ

provide state information on for instance velocity, location and accelerations. The state in reinforcement learning can also be a bit more flexible and abstract: they can be as high or as low level as you want. States can include raw sensor data (low level) or shapes of objects in a room (high level). Also the actions are less conventional. In classic control situations actions can for instance be a low level value motor voltage. In reinforcement learning actions can be low level, just as the classic control situation, but they can also be more high level: 'open the door', 'walk to point A' or 'have lunch'. These actions are described by the book 'Reinforcement Learning: an introduction' by Sutton and Barto [37]. Video games like PacMan have discrete action spaces: the number of different actions that can be taken are finite. In the game of PacMan, you control a yellow agent that has to escape the ghosts and collect round reward balls in the process. You control the PacMan through a maze using one of four actions: "up,down,left,right" (figure 2.3). Real life robotics tend to have a continuous action space. The choice

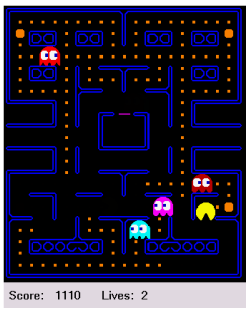


Figure 2.3: PacMan is an example of a discrete action space, with four actions to choose from

of action space has consequence for the type of algorithm applied in a later stage.
The book by Sutton and Barto also talks about the boundary between agent and environment. In robotics it is easy to assume that the boundary between agent and environment is the physical boundary. In the case of a drone, the agent would be the entire drone and the environment would be the rest. However, the agent-environment boundary could be placed somewhere else. Everything that can't be changed or influenced by the agent is outside of it and thus part of the environment.

2.1.3. Rewards

To analyze the performance of the agent, the agent receives a reward r after each state transition. For a whole trajectory τ, each time step the agent will receive a reward $r_t, r_{t+1}, r_{t+2}, ..., r_{t_n}$. The accumulation of all those rewards is:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3}..... \tag{2.3}$$

Where G_t is called the *return*: the summed reward over all the time steps from $t = 0$ until $t = \infty$ in case of infinite time horizon or $t = \tau$ in case of a finite trajectory. This value of return can be used

recursively:

$$G_t = R_{t+1} + G_{t+1} \tag{2.4}$$

Future rewards can also be discounted by a factor γ and is called the *discount factor*, where $\gamma = [0, 1)$, see equation 2.5.

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{2.5}$$

This discount factor determines how important the immediate reward is with respect to the rewards in a later time-step. Intuitively this can be explained by: Do i want my reward right now, or later? Mathematically this γ is important because non discounted reward functions might not converge to a finite value. The goal of the agent is to maximize this cumulative reward. The formulation of this reward function is essentially determining the actual goal of the agent. The implementation of this reward function is an essential and distinctive feature of reinforcement learning. This reward makes sure that the agent learns what actions are good and what actions are bad. Similar to the baby that learns how to crawl, the reward is what makes the baby learn good behavior.
The reward functions are highly flexible and can be as simple or as difficult as the developer wants it. One of the most simple reinforcement learning problems is the cart-pole problem as described in previous section. In this problem, a reward of +1 could be given for not falling. Not falling would be defined as x and θ being lower than a certain threshold value. In a robot navigation task, the robot would score +1 for arriving at the goal, but -1 for each time step: the robot is punished with time so it will try to reach the goal as fast as possible. In the games Go or chess, sometimes seemingly bad states *(or sacrifices)* need to be made in order to later on win the game. There are different road that lead to Rome, and multiple tactics could be used to achieve the goal. This makes it important to tell the agent via the reward function *what* you want to achieve, but not *how* you want to achieve it. Putting too much of the *how* into the reward function leads to bias.

2.1.4. Bellman equation and value functions

In RL, the agent will try to maximize the cumulative reward, or the *return* (equation 2.5). The notion of 'how good' a state is, is the expected return. This value is calculated with means of a *value function*. The value of a state *s* is the expected return of an agent when in state *s* and following the policy π. For an MDP, this captured in the *state-value function V*, equation 2.6.

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s_t] = \mathbb{E}_\pi[\sum_{k=0}^{k=\infty} \gamma^k r_{t_k}|s] \tag{2.6}$$

The reward for a terminal state is always zero. Aside from the state-value function $V_\pi(s)$, we also have the *action-value function* Q. This action-value function defines the value of being in a certain state s, taking an action a and then following the policy π thereafter, see equation 2.7.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t, a_t] = \mathbb{E}_\pi[\sum_{k=0}^{k=\infty} \gamma^k r_{t_k}|s_t, a_t] \tag{2.7}$$

Solving an MDP problem this way means finding a policy that is better than other policies: the optimal policy π^*. It is possible to have multiple optimal policies, but they all share the same optimal state-value function $V^*(s)$, defined as:

$$V^*(s) = \max_\pi V_\pi(s) \tag{2.8}$$

The optimal policies also share the same action-value function $Q^*(s, a)$:

$$Q^*(s, a) = \max_\pi Q_\pi(s, a) \tag{2.9}$$

The optimal action-value function Q^* is taking an action a while being in state s, and following the optimal policy π^* thereafter. Writing Q^* in terms of V^* results in:

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma V^*(s_{t+1}|s_t, a_t)] \tag{2.10}$$

2.1.5. Q-Learning

Q-learning is an off-policy temporal difference algorithm designed to find the optimal Q-function (and with it, an optimal policy). The difference between an off-policy and on-policy strategy is that an off-policy strategy updates its Q-values using the next state s' and *greedy* action a', even though the policy is not taking the greedy action. An on-policy strategy would update the Q-values according to the exploration action a''

Q-learning starts with chosen values for Q, but is updated using an algorithm with at the heart a *Bellman equation*. The Q-learning algorithm as described by Sutton and Barto [37] is depicted in figure 2.4.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 2.4: Q-learning algorithm

At first the Q values are initialized. A main loop is started. An action is chosen under current policy. The Q value is then updated via the update rule. The update rule uses temporal difference learning and bootstrapping: it is updating itself each time-step using the difference between a target Q value and the real Q value, see figure 2.5. The learning rate is comparable to a step size in regular control

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \Big)}^{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$

Figure 2.5: Q-learning update rule

theory. This learning rate is usually chosen in the order of magnitude of $1e-5$. The discount factor is the same as we know from the reward function. A discount factor of 0 will make the agent short sighted and will only look at the current reward. Using a value of 1 will cause the history to be infinitely long. Often used value of the discount factor usually are between 0.9 and 0.99.
The target Q value in the update rule contains a maximization over the actions, which causes a significant positive bias of the target Q value. This problem of overestimation can partially be solved by using *Double Q-learning*. This algorithm uses a second Q-function. The second Q-function will be used as a target for updating the first one. Each update, either the first Q-function is updated using the second one as target, or vice versa: the second Q-function is updated using the first Q-function as target. At each time-step, the change of updating the first or second Q-function is 0.5. This algorithm is depicted in figure 2.6.

2.1.6. Experience replay and frozen target network in Q-learning

Deep Q-learning, or DQL is the implementation of neural networks as universal approximators of the Q-functions. A disadvantage of using this structure in Q-learning is the correlation in temporal difference learning that tends to overfit the networks during training. A widely used solution for this is *experience replay* [27]. The experience replay method saves the agents experiences $e = s_t, a_t, r_t, s_{t+1}$ in a replay

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q_\cdot(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Figure 2.6: Double Q-learning algorithm

buffer. During the application of the Q-value updates, a minibatch is randomly sampled from this replay buffer. This has two advantages:

1. No temporal correlation in training
2. Higher data efficiency

Because the data is randomly sampled from many different episodes, the temporal correlation that tends to overfit the neural networks is no longer an issue. The method is also more data efficient since one experience can potentially be used in many weight updates.
Freezing the target network means that the target network is not updates as frequently as the actual Q-function. Every k episodes (where k is a user defined parameters) the network parameters from the actual Q-function are copied to the target network. This helps stabilizing training since the actual Q-value has more time-steps to approximate the target network value. DQL has a discrete output and cannot be used on continuous action spaces [14].

2.1.7. Policy gradient algorithms

Q-learning (and its derivative methods) are a proper method to solve reinforcement learning problems. By learning values of the possible actions Q-learning makes it possible to select the actions with the highest value, and thus find the best possible policy. Policy gradient methods are different: they learn a parameterized policy that does not need to consult a value function. A value function can still be used to update the parameter (DDPG and TD3 algorithms, to be discussed later), but the policy will not use the value function to choose an action.
The policy now is dependent on its parameters θ, for which the notation is $\pi(a|s, \theta)$. The performance of the policy is measured by some value $J(\theta)$. Updating the policy parameters is done via the parameter update rule (equation 2.11.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.11}$$

Where $\nabla J(\theta_t)$ is the gradient of the performance with respect to the parameter vector θ and α is again the learning rate. The policy can be parameterized in any way as long as the policy is differentiable with respect to its parameters, that is if the gradient of the policy with respect to its parameters exists and is finite.

2.2. Actor-critic methods

Actor-critic (AC)methods use both value functions as well as policy gradient methods to obtain a policy. The actor refers to the policy: the one taking the actions. The critic refers to the value function that judges the actions and states of the policy. The advantage of using an actor rather than pure policy gradient methods is that the policy gradient methods use sampling of the true gradient, which could

cause gradients to point in a wrong direction. Actor-Critic methods use the critic to reduce the variance of the gradient of the actor. A vanilla AC algorithm is depicted in figure 2.7 [8].

Algorithm Vanilla Actor Critic

while not converged **do**
take action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
fit $V_\phi^\pi(s)$ using target $r + \gamma V_\phi^\pi(s')$
evaluate $A^\pi(s, a) = r(s, a) + \gamma V_\phi^\pi(s') - V_\phi^\pi(s)$
$\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \ln \pi_\theta(a_i|s_i) A^\pi(s_i, a_i)$
$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end while

Figure 2.7: Vanilla Actor-Critic algorithm

In this algorithm it is visible that the value function is used to judge the action and to calculate the estimation of the gradient of the cost $\nabla_\theta J(\theta)$. AC methods have shown promising result. Due to their applicability on *continuous actions spaces* they are widely used in the field of robotics, where the nature of the platform often has a continuous action-space rather than a discrete one.

2.2.1. DDPG

Deep Deterministic Policy Gradient fuses the success of DQL with deterministic policy gradient to create a model free algorithm which can be used on continuous action and state-spaces [21] [27]. DDPG is a widely used method to solve continuous problems such as *continuous lunar lander* and other open-AI gym environments [1]. In previous policy gradient method, the policy is always modeled as a distribution over possible actions depending on a certain state and is thus *stochastic* in nature. This method is favored over discretizing continuous action and state spaces because of the curse of dimensionality: "*the number of possible actions increases exponentially with the number of degrees of freedom*" [21]. DDPG uses an Actor and a Critic network, as well as a target Actor and target Critic network (in total four networks). The actor network $\mu(s|\theta^\mu)$ determinalisticly maps states to actions. The actor is updated using the critic for gradient calculation, just as in regular AC methods. However, this method still has some of the same issues as normal DQL methods: Temporal correlation is still an issue. Therefore DDPG also makes use of the Replay Buffer. Also the instability due to the use of nonlinear function approximators (deep neural networks) is an issue. The solution is the use of the target networks (first introduced by Mnih et al. [26]) to stabilize learning.
Because the nature of the actor is deterministic, the exploration-exploitation dilemma needs a solution. This is solved by creating an exploration policy μ' which is the actor policy, but with added noise. This noise can be chosen to suit the environment.

$$\mu'(s_t) = \mu(s_t) + \mathcal{N} \tag{2.12}$$

The complete DDPG algorithm is depicted in figure 2.8.

2.2.2. TD3

Twin Delayed Deep Deterministic Policy Gradient (TD3) is an upgraded version of the DDPG algorithm developed in the paper *Addressing Function Approximation Error in Actor-Critic Methods* by Fujimoto et al. [9]. The paper proposed methods to decrease overestimation of value functions due to function approximation errors using three tricks:

1) Clipped Double Q-learning: This trick is similar to Double Q-learning, explored in subsection 2.1.5. TD3 learns a second Q-function. In the Bellman loss function, the value of both Q-functions is compared, and the lowest of the two is chosen.
2) Delayed policy updates: The actor (policy) and the target networks are updated less frequently then the Q-function. This gives the Q-function more time to reduce estimation error thus decreasing variance by having more time to converge.
3) Target policy smoothing: TD3 adds noise to the target action. This causes a certain amount of

[1] open-AI gym is a widely used RL environments library: gym.openai.com

Algorithm DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 2.8: DDPG algorithm

randomness in the action, preventing overestimation of the Q-function values to accumulate.

The full algorithm of TD3 is depicted in figure 2.9.

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ
with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$
    $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
  end if
end for

Figure 2.9: TD3 algorithm

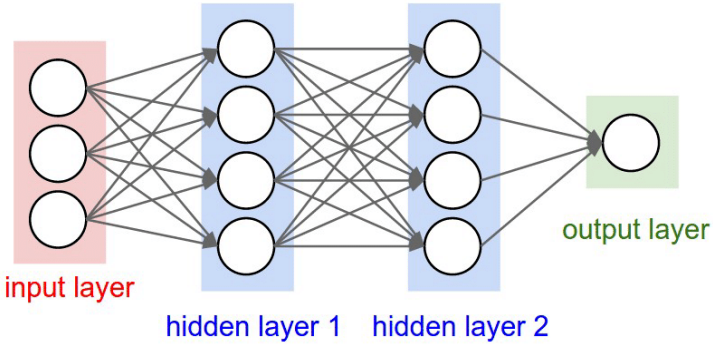


Figure 2.10: schematic of a simple DNN with 2 hidden layers

2.3. Deep Neural Networks

Deep Neural Networks (DNN) play a big roll in Reinforcement Learning. DNN and RL combined is often called Deep Reinforcement Learning (DRL). A deep neural network is a non-linear function approximator. DNN have a variety of uses, including speech recognition, image recognition and robot control [8]. Via *backpropagation*, DNN's can adjust the weights of the network to adjust the network output to copy the intended output. This is used in image recognition for example. Labelled datasets with known output are used to train the neural networks to give the correct output when the image is given as input. A DNN consists of at least (but not limited to) an input layer, two or more hidden layers, one output layer. These layers consist of one or more nodes. All of the nodes in one layer are connected to all the nodes in the next layer with arrows called 'weights'. The architecture of simple DNN depicted in figure 2.10 [2].

2.3.1. Forward propagation

Forward propagation is the process of computing the output of a neural network by a series of matrix multiplications. To compute the next layer in figure 2.10 is equivalent to a linear matrix multiplication: Hidden layer 1 is a dot product of the input layer with the weights. If the input was a 1x3 vector $[i_1 i_2 i_3]$ and the weights are depicted as a 3x4 matrix w_{ij}, the nodes of the first hidden layer can be computed by:

$$\begin{bmatrix} i_1 & i_2 & i_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix} = \begin{bmatrix} hl_1 & hl_2 & hl_3 & hl_4 \end{bmatrix} \tag{2.13}$$

This process can be repeated until the output layer is calculated. These linear matrix multiplications are not yet capable of approximating non-linear functions. To add the non-linear functionality to the DNN's, activation layers can be added.

2.3.2. Activation layers

Activation layers are non-linear operations performed on the values of the nodes. Any non-linear function can be used as activation layer. In current DNN's, the most used activation functions are the *ReLu* (rectified linear unit), *Sigmoid* and *Tanh* (hyperbolic tangent). A ReLu layer will bound the output to have positive values. Tanh layers are used when an output needs to be bounded between (-1,1).

2.3.3. Backpropagation

Backpropagation is essential in computing the weights of a network. Backpropagation computes the gradients of the weights and biases using stochastic gradient descent to update the parameters in a direction that minimizes a loss function:

$$\theta = \theta - \alpha \nabla_\theta L(\theta) \tag{2.14}$$

Where L is the loss function to be minimized. In current algorithms a variant of stochastic gradient descent is used, called ADAM optimizer. ADAM varies the step size α depending on the data distribution.

3

quadcopter dynamics and classic control

This chapter will explain the basic working principle of a quadcopter. The dynamic and kinematic equations of the quadcopter are analysed using the Newton Euler equations for a rigid body. Pixhawk[1], Ardupilot and most on the market flight controllers use a cascaded loop PID controller for attitude and position control. This type of controller will be explained and dissected to understand the current most used version of quadcopter control.

3.1. Basic principles

A quadcopter (or quadrotor, drone) is an unmanned aerial vehicle (UAV) that generates lift and thrust using four rotors. The body holds the electronics of the quadcopter and is usually placed in the centre. The four motors are attached to the body on arms. The brains of the quadcopter is called the *flight controller*. The flight controller accepts all the sensor data present. These sensors usually include a gyroscope and accelerometer, but can be extended by for instance GPS, optical flow sensors, Lidar range sensors, barometer and many more. The type and amount of sensors depend on the functionality of the drone. If a drone is flown manually, a pilot uses a controller to give input to the drone. Autonomous drones use more information about the surroundings to determine its path. This information can be gathered by distance sensors , but might also be achieved by using some sort of vision system (camera). Depending on the level of autonomy, the amount of sensors will vary. The flight controller uses the information given by the sensors (and if flown manually: the control stick input) to determine the output it will send to the motors. This output is routed to an electronic speed controller (ESC). The ESC will translate the output from the flight controller to a usable signal used by the brushless DC voltage motors commonly used in quadcopter flight. More information about the anatomy of a drone is discussed in chapter 4.

3.1.1. Coördinate frames

Two coördinate frames are used when talking about quadcopter systems. First, there is the *inertial frame* (or world coördinate frame). This coördinate frame lives in Euclidean space where the Newtons laws are valid. The inertial frame is a right handed coördinate system, with the option of one of two conventions. One option is the *"east-north-up" (ENU)* convention, where the positive x axis points to the right (east), the positive y axis points forward (north) and the positive z axis points upward (up), see figure 3.1. The second option is with *"north-east-down" (NED)* convention, where the positive x axis points forward (north), positive y axis points right (east) and positive z axis points down (down), see figure 3.2

The other coördinate frame used is the body-fixed frame (or body frame). This coördinate frame is assumed to have its origin in the centre of mass of the quadcopter and follows the NED convention. Forward direction is aligned with the x axis, y axis points to the right side of the quadcopter and positive z axis points down, see figure 3.3 In mechanical engineering the ENU convention is most used. However,

[1] widely used open source flight controller (www.pixhawk.org)

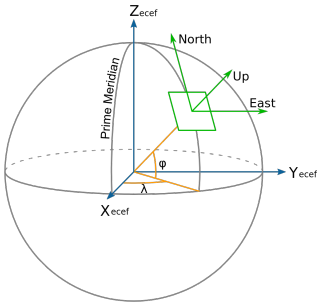


Figure 3.1: ENU coördinate system

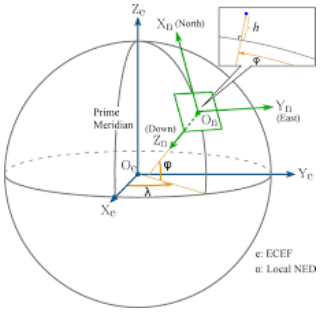


Figure 3.2: NED coördinate system

in aerospace engineering the NED convention is most widely used. Therefore the NED convention is used throughout the rest of the thesis.

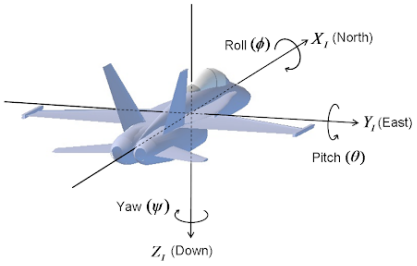
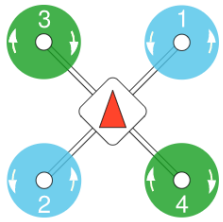


Figure 3.3: Body frame coördinate system

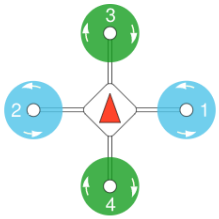
In this body frame also the motions roll, pitch and yaw are defined according to aerospace conventions. Roll (ϕ) is the rotation around the x-axis, pitch (θ) is the rotation around the y-axis and yaw (ψ) is the rotation around the z-axis. Note that this coördinate system is a right handed system and the positive directions of the rotations are also the right handed rotations.

3.1.2. Airframes

A quadcopter can have different airframes. An airframe is the configuration of components, for instance the length of the motor arms to the center of gravity, the direction of the x-axis with respect to the rotors and the angle of the motor arms with respect to the x and y axes. The Pixhawk website lists multiple supported airframes. We assume for this thesis a symmetric quadcopter, where all the motors are evenly spaced out and with the same distance to the origin. With that assumption, there are two main types of quadcopter airframes: the '+' and 'x' configurations. These configurations are depicted in figure 3.4a and 3.4b. In these figures, the red arrow points in the forward direction aligned with the x axis. The numbers on the rotors indicate the convention for rotor numbering in the two airframes. The arrows on the rotors indicate the rotation of the rotors.



(a) x configuration



(b) + configuration

3.2. Flying the drone

The drone flies by the spinning rotors like a helicopter, generating lift. The total lift is the summation of the lift generated by each rotor. Hover is achieved if the attitude of the quadcopter is zero for pitch and roll and if the total amount of generated lift is equal to the gravitational force pulling the drone down. Increase in altitude is achieved by generating a total lift more than the gravitational force, while a decrease in altitude is achieved by generating a total lift less than the gravitational force. The pitch, roll and yaw movements are achieved by differences in force generated on each side of the respective axis. Depending on the airframe this will result in different motor behavior.

3.2.1. x configuration

Pitch is achieved by increasing the difference in thrust of rotor set (1 3) with respect to rotor set (2 4). Roll is achieved by creating a difference in of rotor set (1 4) with respect to rotor set (2 3). Yaw is achieved by creating a difference in thrust between the rotors rotating in clockwise and the rotors rotating in counter-clockwise direction, so a difference between thrust of rotor set (1 2) with respect to rotor set (3 4).

3.2.2. + configuration

Positive pitch is achieved by increasing the difference in thrust of rotor 3 with respect to rotor 4. Roll is achieved by creating a difference in of rotor 1with respect to rotor 2. Yaw is achieved by creating a difference in thrust between the rotors rotating in clockwise and the rotors rotating in counter-clockwise direction, so a difference between thrust of rotor set (1 2) with respect to rotor set (3 4).

3.3. Rotation and Transformation

The transformation from body frame to inertial frame is expressed as the 'rotation matrix' $R_{zyx}(\phi,\theta,\psi) \in SO3$. The rotation matrix $R_{zyx}(\phi,\theta,\psi) \in SO3$ (equation 3.1) is computed by multiplying the rotation matrices of each of the individual axis [34] where to rotations to each axes correspond with the matrices in equations B.1, B.2, B.3, see appendix B.

$$R = \begin{bmatrix} cos\theta cos\phi & sin\psi sin\theta cos\phi - cos\psi sin\phi & cos\psi sin\theta cos\phi + sin\psi sin\phi \\ cos\theta sin\phi & sin\psi sin\theta sin\phi + cos\psi cos\phi & cos\psi sin\theta sin\phi - sin\psi cos\phi \\ -sin\theta & sin\psi cos\theta & cos\psi cos\theta \end{bmatrix} \tag{3.1}$$

A rotation from body to inertial frame is described by the rotation matrix R. A translation from body to inertial frame is described by the linear transformation matrix T (equation 3.2)[20].

$$T = \begin{bmatrix} 1 & sin\phi tan\theta & cos\phi tan\theta \\ 0 & cos\phi & -sin\phi \\ 0 & \frac{sin\phi}{cos\theta} & \frac{cos\phi}{cos\theta} \end{bmatrix} \tag{3.2}$$

To rotate or translate in the opposite direction, from inertial to body frame, the transpose of the rotation and translation matrices are used: R^T and T^T.

3.4. Mathematics and Dynamics model

Assumptions made while developing this dynamics model are [3]:

- The quadcopter is a rigid body
- Centre of mass and centre of gravity coincide with the geometrical centre
- Moments of inertia of the propellers are neglected
- No sideslip during flight
- No bladeflapping
- No aerodynamic interaction between blades
- No ground effect
- Low flight speeds of under 10 [m/s]

This section will explain more about the underlying principles of these movement in body and inertial frame after which the dynamics equations of these motions are set up. The general Newton-Euler equations are set up [10] [39]. The different subsections within these general equations are then further specified.
First of all, we define the linear and angular position in inertial frame as vector $[x\ y\ z\ \phi\ \theta\ \psi]^T$. The linear velocity and angular velocities are defined in the body frame as vector $[u\ v\ w\ p\ q\ r]^T$. u, v and w correspond to respectively forward, sideways and downward motion of the rigid body. p, q and r correspond to respectively roll, pitch and yaw speeds. Switching between speeds in inertial and body frame uses the rotation matrix R from equation 3.1.

$$v = Rv_b \tag{3.3a}$$

$$\omega = T\omega_b \tag{3.3b}$$

Where the underscore b in v_b and ω_b stands for the speeds in body frame. $v = [\dot{x}\quad \dot{y}\quad \dot{z}]^T$, $\omega = [\dot{\phi}\quad \dot{\theta}\quad \dot{\psi}]^T$, $v_b = [u\quad v\quad w]^T$ and $\omega_b = [p\quad q\quad r]^T$. Equations 3.3 combined make up the kinematic equations of the quadcopter.

3.4.1. Newton-Euler equations

Given the coördinate systems described in the beginning of this section, the full generalized non-linear Newton-Euler description of the quadcopter is given in equation 3.4 [1].

$$F_b = m(\omega_b \times v_b + \dot{v}_b) \tag{3.4a}$$

$$M_b = I\dot{\omega}_b + \omega_b \times (I\omega_b) \tag{3.4b}$$

In this equation, $F_b = [F_x\quad F_y\quad f_z]^T$ represents all forces on the rigid body quadcopter. m is the mass of the quadcopter.
$M_b = [M_x\quad M_y\quad M_z]^T$ are all the moments acting on the quadcopter, where I is the diagonal inertia matrix

$$I = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \tag{3.5}$$

The total forces and moments can be further specified by adding prior knowledge to the equations. Together, these equations make up the dynamic equations of the quadcopter, see equation B.4.

3.4.2. Forces and moments

In previous subsection, the general Newton-Euler equations are specified, resulting in general dynamics equations B.4. The right hand side of these equations consist of system states and system inherent properties mass and inertia. The left hand side of these equations are all the forces and moments acting on the rigid body. These forces and moments exist due to gravity, thrust and aerodynamic effects [17]. In this subsection, the influence of each of these factors is determined and added to the dynamics equation. The total force can be rewritten in its components, see equation 3.6.

$$F_b = F_g + F_t + F_a \tag{3.6}$$

Where F_g is the force induced by gravity, F_t is the force generated by thrust, and F_a incorporates the aerodynamic drag effects on the quadcopter. In the same way, the moments acting on the body can be split up, see equation 3.7.

$$M_b = M_c + M_a + M_g \tag{3.7}$$

Where $M_c = [\tau_x \quad \tau_y \quad \tau_z]$ are the generated control torques by the difference in rotor speeds, $M_a = [\tau_{wx} \quad \tau_{wy} \quad \tau_{wz}]$ are the torques caused by aerodynamic drag forces and M_g are torques induced by gyroscopic effects of the rotor blades.

Gravity only has an effect in the z-direction of the inertia frame. This results in equation 3.8.

$$F_g = mgR\hat{e}_z \tag{3.8}$$

Where m is the mass, g is the gravity constant, and $\hat{e}_z$ is the unit vector in the inertial z-direction. F_t is the force acting on the rigid body caused by the thrust (or thrust) of the rotors. Thrust acts on the centre of gravity in the direction of the negative z-direction in body frame, $\hat{e}_3$.
Equation 3.6 can be elaborated to equation 3.9.

$$F_b = mgR\hat{e}_z + F_t\hat{e}_3 + F_w \tag{3.9}$$

Adding the forces and moments to the dynamics equations by substituting the equations of the forces into the dynamics equations B.4, equation B.5 is obtained.

3.4.3. Actuator dynamics

Equation B.5 is already a more detailed formula regarding the system dynamics. In the introduction of this chapter is described that a drone flies because of thrust generated by spinning the rotors. Differences in rotor speeds of the four rotors cause torques to act on the body. This thrust force and these torques are described by actuator dynamics. After identifying these dynamics, they can be implemented in the system dynamics equations.
The thrust is the summation of the individual thrusts generated by each of the rotors. A rotor generates force proportional to the square of the rotor speed, with some thrust coefficient. The sum of all the four rotor forces make up the thrust force F_t (equation 3.10).

$$F_t = \sum_{i=1}^{4} c_t \Omega_i^2 \tag{3.10}$$

In this equation c_t is the thrust coefficient of the rotor (which is assumed to be equal for each rotor) and Ω is the rotor speed (in rad/s) for each of the rotors. The part of the actuator dynamics that is responsible for the generated torques is the part where quadcopter configuration makes an important difference. For a + configuration, the roll, pitch and yaw torques are depicted in equation 3.11.

$$\begin{cases} \tau_x = c_t l(\Omega_2^2 - \Omega_1^2) \\ \tau_y = c_t l(\Omega_3^2 - \Omega_4^2) \\ \tau_z = c_\tau l(-\Omega_1^2 - \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \end{cases} \tag{3.11}$$

Where c_t is the thrust coefficient of the rotor and c_τ is the torque coefficient of the rotor. The length of the rotors to the axis is depicted as l. Since the quadcopter configuration is assumed to have symmetry in both x and y axis, and the all distances are equal, l is the same for each of the rotors. For x configuration, the roll, pitch and yaw torques are depicted in equation 3.12.

$$\begin{cases} \tau_x = c_t l(\Omega_2^2 + \Omega_2^2 - \Omega_1^2 - \Omega_4^2) \\ \tau_y = c_t l(\Omega_1^2 + \Omega_3^2 - \Omega_2^2 - \Omega_4^2) \\ \tau_z = c_\tau l(-\Omega_1^2 - \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \end{cases} \tag{3.12}$$

If chosen a state vector $x = [x \quad y \quad z \quad u \quad v \quad w \quad \phi \quad \theta \quad \psi \quad p \quad q \quad r]^T$, from equations B.5 and 3.3, the system can be represented as equation 3.13:

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{u} \\ \dot{v} \\ \dot{w} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} w(sin\phi sin\psi + cos\phi cos\psi sin\theta) - v(cos\phi sin\psi - cos\psi sin\phi sin\theta) + u(cos\psi cos\theta) \\ v(cos\phi cos\psi + sin\phi sin\psi sin\theta) - w(cos\psi sin\phi - cos\phi sin\psi sin\theta) + u(sin\psi cos\theta) \\ w(cos\phi cos\theta) - u(sin\theta) + v(cos\theta sin\phi) \\ rv - qw - g(sin\theta) + \frac{f_{wx}}{m} \\ pw - ru - g(sin\phi cos\theta) + \frac{f_{wy}}{m} \\ qu - pv + g(cos\theta cos\phi) + \frac{f_{wz} - F_t}{m} \\ p + r(cos\phi tan\theta) + q(sin\phi tan\theta) \\ q(cos\phi) - r(sin\phi) \\ r\frac{cos\phi}{cos\theta} + q\frac{sin(\phi)}{cos\theta} \\ \frac{I_y - I_z}{I_x} rq + \frac{\tau_x + \tau_{wx}}{I_x} \\ \frac{I_z - I_x}{I_y} pr + \frac{\tau_y + \tau_{wy}}{I_y} \\ \frac{I_x - I_y}{I_z} pq + \frac{\tau_z + \tau_{wz}}{I_z} \end{bmatrix} \tag{3.13}$$

This equation can be captured in simplified form (equation 3.14).

$$\dot{x} = f(x, u) \tag{3.14}$$

Where f is a (nonlinear) function of the state and the inputs.

3.4.4. Linearized model

Linearization is often used to develop controllers (PID control) for Linear Time Invariant (LTI) systems [16]. LTI systems are linear. Since the dynamics equations of the quadcopter are highly nonlinear, linearization is used to develop controllers for LTI systems [32] [31] [29]. Linearization of a system creates the opportunity to analyse a non-linear system around one operation point. A linear approximation is made around that operation point. Angles are assumed small around this operation point, so that according to the small-angle approximation $sin\theta = \theta$, $cos\theta = 1, tan\theta = \theta$ [40]. This small angle assumption applied to equation 3.13 results in equation B.6.

As operation point for linearization, usually hover state is used [32]. Applying both the small angle assumptions and the hover state, the linear model is given [25], see figure B.7.

3.5. Quadcopter control

Since linear control imposes the linear boundaries to the system, and therefore limits the movements of the quadcopter, the model used throughout the rest of the thesis is the non-linear model as described in equation 3.13.

In existing flight controllers multiple flight control modes exist, each using their own control diagram. The PX4 documentation shows a list of their supported flight modes [2], as of early 2020 including *manual/stabilized mode* and *acro mode* in the 'manual flight' subsection, *attitude mode* and *position mode* in the 'assisted flight' subsection and various totally autonomous flight modes in the 'autonomous flight' subsection. The manual flight modes translate the control stick input to either roll, pitch and yaw set-points for the controller, or roll, pitch and yaw *rate* set-points. This type of control is more low level than the one used in autonomous flight, where via a pre-planned mission (or in real time updated) location set-points, or even higher level mission targets are put into the controller.

In this thesis a totally autonomous quadcopter accepting 3D position target is assumed. This autonomous quadcopter accepts multiple way-points in the form of a trajectory as input. The used control

[2] https://dev.px4.io/master/en/concept/flight_modes.html

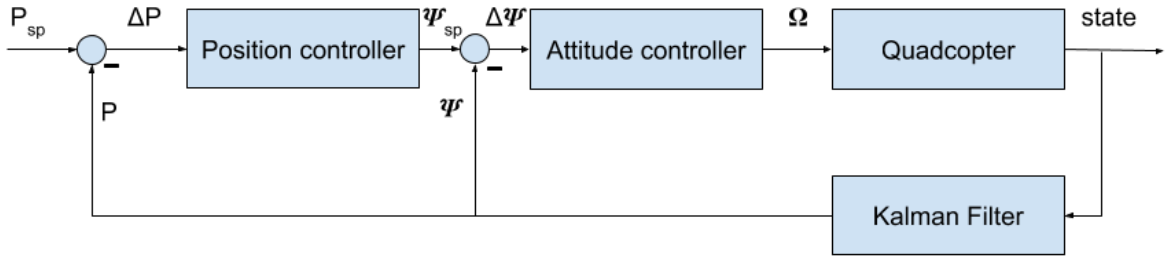


Figure 3.5: Block schematic of quadcopter PID control loop

method for a position control system is a PID control loop, existing of an attitude control block and a block for position control, see figure 3.5.

A position set-point r_{sp} is compared to the actual position of the drone $\hat{r}$. The difference Δr enters the position controller block. This block computes the desired acceleration and the desired attitude using Proportional Derivative Integral (PID) control, see equations 3.15 and 3.17.

$$Acc_{des} = K_p(z_{sp} - z) + K_d(\dot{z}_{sp} - \dot{z}) \tag{3.15}$$

Of which Acc_{des} is vector with three columns: x, y and z acceleration. The desired z acceleration is used to calculate the desired thrust (equation 3.16), while the desired x and y acceleration are used to calculate the desired pitch and roll angle. The desired thrust is computed by taking the difference in desired thrust with and gravity and multiplying by the mass m.

$$F_{thrust} = m * (g - \ddot{z}) \tag{3.16}$$

$$\phi_{des} = \frac{1}{g}(\ddot{x}sin(\psi_{des}) + \ddot{y}cos(\psi_{des})) \tag{3.17a}$$

$$\theta_{des} = \frac{1}{g}(-\ddot{x}cos(\psi_{des}) + \ddot{y}sin(\psi_{des})) \tag{3.17b}$$

In these equations, g is the gravitational constant. The desired roll and pitch angle enter the attitude control block. The desired torques are calculated via PD control, as seen in equation 3.18.

$$\tau_{xdes} = K_{p\tau}(\phi_{des} - \phi) + K_{d\tau}(\dot{\phi}_{des} - \dot{\phi}) \tag{3.18a}$$

$$\tau_{ydes} = K_{p\tau}(\theta_{des} - \theta) + K_{d\tau}(\dot{\theta}_{des} - \dot{\theta}) \tag{3.18b}$$

From the actuator dynamics equations 3.10 and 3.12, the relations between forces moments and motor outputs is evident. Extracting the motor speeds from these equations and putting the desired thrust force and the desired torques in a joint vector (equation B.8), the motor speeds can be extracted by simple matrix multiplication, see equation 3.19:

$$\begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix} = \begin{bmatrix} k_t & k_t & k_t & k_t \\ -lk_t & lk_t & lk_t & -lk_t \\ lk_t & -lk_t & lk_t & -lk_t \\ -k_\tau & -k_\tau & k_\tau & k_\tau \end{bmatrix}^{-1} \begin{bmatrix} F_{thrust} \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} \tag{3.19}$$

3.6. Conclusion

This chapter discusses the working principles of a quadcopter and analyses the non-linear dynamics. Understanding these two items is essential in creating a simulation model. This simulation model is used by the RL environment to compute the states of the agent, given a certain state and input. The accuracy of this simulation model is an important factor in the sim-to-real transfer. The controller is used in testing the two-step parameter estimation procedure in chapter 5.

4

Custom quadcopter build and Identification

4.1. Introduction

Chapter 3 discusses the working principles of a quadcopter, as well as reviewing its dynamics. For this research a custom quadcopter is built. This quadcopter is the platform that will be controlled by the Agent and will function as the testing platform for this thesis. Off the shelf quadcopters are closed platforms: It is difficult (or impossible) to alter anything about the software or hardware. For experimental platforms therefore often is chosen for a modular building platform, with an alterable flight controller. PX4 and Arducopter (amongst other brands) provide flight controllers that can interact with third party hardware (motors, ESC's and such). Adaptability with these flight controllers is higher than off the shelf quadcopters. Where off the shelf total quadcopters don't allow for any modifications, these flight controllers are open source: the source code is found online and can be adapted. This allows for modifications regarding the sensors, the frame and electronics used. The code can also be altered, so different modules can be added or removed from the flight controller. For the system identification of this quadcopter, we will use an RC controlled platform, with sensors that can capture the state of the drone. PixHawk does provide these options. The **first build** will therefore use a PixHawk flight controller, running PX4's latest build *px4_v5* to gather data for identification of parameters and to verify the feasibility of the rest of the components. However, as of today, these flight controllers don't offer the option of running python code. The eventual RL agent will run as a python file. A **second build** will be created, running on a Raspberri Pi 3B+ [1]. The Raspberry Pi (RPi) is a small linux based computer and will run the python script that will obtain sensory data and send motor command to the hardware. Also a third build will be implemented, using a three axis free rotation setup. This setup is used for save testing of attitude control algorithms. The drone is hung into this setup, bounding it position wise, but enabling rotation around all three axes.

4.2. Design requirements

To create a quadcopter from scratch, a list of requirements is needed. This quadcopter will be used as a platform to test flight control algorithms, in collaboration with engineering company *Fusion Engineering* [2]. The experimental nature of this platform does not guarantee safe flight performance and is likely to endure crashes. Therefore the first design requirement is that the quadcopter needs to be *crash proof*. The drone also might be used as platform for testing different sensor sets, for instance multiple GPS sensors, Lidar range sensors, a camera and optical flow sensors. To make sure all of these sensors can be mounted, the quadcopter has a maximum mass of maximum 3 kg. The motors need to produce *High thrust*. Quadcopters are inherently unstable platforms. When no control input is given to the system, the state values will grow out of bounds. To minimize the instability of the system, a *high inertia* is needed. This will slow down the rotational motion of the quadcopter. Since the research is

[1] Raspberri Pi is a small computer: www.raspberrypi.org
[2] www.fusion.engineering

about a quadcopter, the platform should at least support four motors.
To summarize, the quadcopter will need to:

1. be crash proof
2. high thrust motor
3. have high inertia

4.3. Components

The basis of each quadcopter consists of multiple components: The frame, the flight controller, the sensors, the motors and the electronic speed controllers (ESC). The frame is the component that functions as the skeleton on which all other components are mounted. The flight controller is the brain of the quadcopter. It receives information from the sensors and computes the signal send to the ESC's. The ESC translates this signal to a three phase AC power that drives the brushless DC motors. A LiPo battery is mounted to power the electronics. The list of components is:

- Frame
  - Frame rods
  - Connection pieces
  - Motor mounts
  - Legs
- Flight controller
- Sensors
  - Accelerometer
  - Gyroscope
  - Vicon Optitrack
  - Barometer
- ESC's
- Motor assembly
  - Motors
  - Rotors
- Battery

4.3.1. Frame

The frame of the quadcopter is the skeleton that holds together all the components. To make the frame as light as possible, the frame is made from square carbon fiber rods with 10 mm sides. A central mounting pad is used in the design to house the components. This central mounting pad consists of a hexagonal laser cut piece of plexiglass. The frame rods are positioned in a square layout, with the diagonals also connected. The diagonal rods are held together by the central mounting pad and an identical piece of plexiglass, where the frame rods are held in between. The corners of the square frame are connected by corner pieces also laser cut from plexiglass. The frame size is chosen so that the motors can fit within the outer square. The anatomy of the frame enables the drone to crash without damaging the motors or the propellers. The motors are placed on the diagonal rods of the frame. A schematic of the drone frame layout can be seen in figure 4.1. In this schematic, the light blue circles represent the rotors, the darker blue circles represent the motors. The motors are connected to the frame rods using 3d printed motor mounts. These motor mounts are created so the distance from the centre can be varied if needed.The frame rods with the motors mounts, motors, rotors and plexiglass connection pieces can be seen in figure 4.2.

The frame should also incorporate legs of any form for the drone to land on.

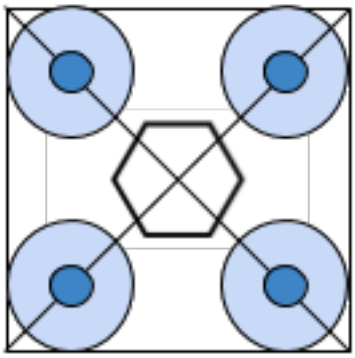


Figure 4.1: Quadcopter design schematic

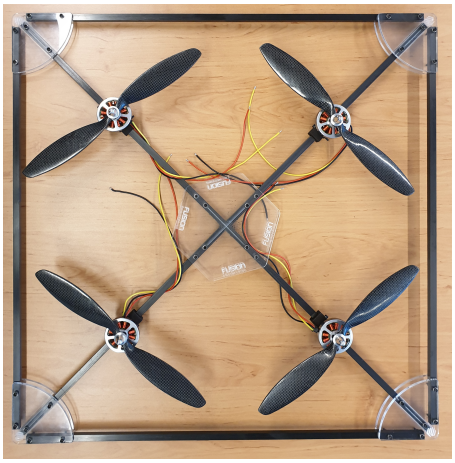


Figure 4.2: Frame rods with plexiglass connections pieces, motor mounts, motors and rotors

4.3.2. Flight Controller

The first build will use a Pixhawk 4 flight controller. The Pixhawk 4 flight controller is bought as is, and is after installation of the other components ready to use. It is shipped with working firmware. Pixhawk 4 flies with a cascaded PID control loop as described in section 3.5. The sensors provide information to the flight control unit. This sensor data is then fused using a Kalman filter. The default Kalman filter used by the pixhawk 4 is their own developed EKF2 filter. [3].
The second build will not use an off the shelf flight controller, but will use a Raspberry Pi 3b+ to directly map sensor data to an output signal for the ESC's.

4.3.3. Sensors

For the first build, the accelerometer and gyroscope used are the built-in accelerometer and gyroscope of the PixHawk. The accelerometer measures linear accelerations. The gyroscope measures rotational velocity. The accelerometer and gyroscope are used for attitude control within the Pixhawk flight controller. GPS is not used since flights only occur indoors. Pixhawk 4 also uses a barometer for altitude measurements. In the first build, external positioning is used to gather visual odometry data for position measurement. The system used is the Vicon [4] system available in the TU Delft cyberzoo [5]. The Vicon system is a motion capture system capable of accurate location tracking. The Vicon system in the cyberzoo uses twelve infrared camera's mounted on the ceiling. Infrared reflecting markers will be positioned on the object of tracking. The motion capture software creates a rigid body from these markers and is then able to determine sub-millimeter precision position and attitude measurements. Where the first build relies on multiple sensors for attitude control and only uses Vicon for position

[3] https://github.com/PX4/Firmware/tree/master/src/modules/ekf2
[4] www.vicon.com
[5] https://nl.wikipedia.org/wiki/Cyber_Zoo

Figure 4.3: Pixhawk 4 flight controller

Figure 4.4: Raspberry Pi 3B+ computer

measurement, the second build uses only the information of the Vicon tracking system (position and orientation) to determine its state and uses this information for control.

4.3.4. ESC

The ESC is responsible for translating the input signal from the flight controller to a three phase AC voltage to send to the brushless DC motors. ESC's are off the shelf components. Multiple protocols exist for communication. The most common one is Pulse Width Modulation, also used by the Pixhawk flight controller. The standard PWM signal is a 50 HZ signal and has a high state duration between 1 and 2 milliseconds. The duration of this high state proportional to the RPM of the motor: a high state duration of 2 milliseconds corresponds to the maximum RPM of the motor, where a high state duration of 1 milliseconds corresponds to the minimum RPM of the motor. A high state duration of 1.5 millisecond will then correspond to half of the maximum RPM of the motor, see figure 4.5. A drawback of using this protocol is the limited refresh rate. The 50 HZ signal is the limit of refreshing motor commands. This protocol is also an analog signal, which makes it vulnerable for interference. Almost all ESC's are capable of accepting PWM signals.
Other than PWM, newer digital protocols are now also used. DShot is one of these newer protocols that is gaining market value nowadays. DShot is the first digital protocol available for quadcopter ESC's. Being a digital protocol, DShot solves the problems that analog protocols have due to interference and communication issues. Also this protocol is faster than its analog counterparts, with the number behind the DShot stating the communication speed: DShot150 can send up to 150.000 bits per second, DShot1200 up to 1.200.000 bits per second. An alternative (or upgrade) to the standard DShot protocol

is the Bidirection DShot protocol [6]. Bidirection DShot allows for communicating the motor speed back to the flight controller, without any extra wires at very high speed. This high frequency feedback is needed for advanced flight controllers in order to react quickly to changes in the environment (eg. wind gusts) or to actively detect faults in the rotors.
The Holybro Tekko32 series ESC's support both Bidirectional DShot as well as PWM and are therefore chosen as an affordable ESC for this quadcopter.

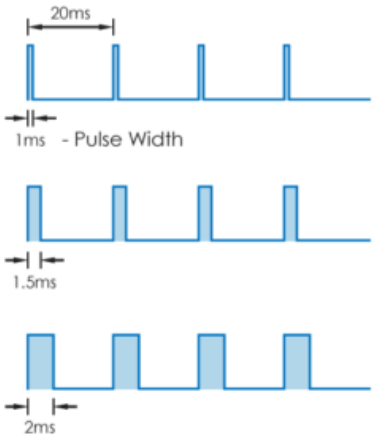


Figure 4.5: PWM signal, varying from 1 ms high (lowest RPM), to 2 ms high (highest RPM)

4.3.5. Motors and rotors

The type of motors is related to the characteristics of the drone. Small drones built for racing require small motors with high KV values. Large drones for lifting require larger motors with low KV values. The KV value of the motor is the rotational speed of a motor per Volt supplied to it under no load. The amount of voltage supplied with brushless DC motors is the output of the battery and is always constant. For our purpose we use a 4S LiPo battery supplying 14.8 V. The DYS d4215-650KV motor is rated to lift 1.5 kg at full speed with a 12", 3.8 pitch propeller (the type of propeller used).

4.3.6. Legs

The legs will often be the first object of impact in case of a crash. The legs can be chosen to be either stiff or elastic. In case of strong and stiff legs, the forces of impact will be passed on to the frame. We want the frame to survive the crash, so the forces to be lower. Therefore a cheap elastic type of leg is used. In case of a crash where the legs break off, the legs will be easy to replace. The legs are made of semi circular PVC tubing, width a width of around 5 cm and a radius of 10 cm.

4.4. Assembly and schematic

4.4.1. First Build: Pixhawk controlled quad

This first build uses a Pixhawk 4 as flightcontroller, see figure 4.6. The pixhawk uses its internal accelerometer, gyroscope and barometer as internal sensors. The sensor data is fused using the Pixhawk extended kalman filter. The Pixhawk sends commands to the ESC's. The raspberry pi is used to wire communication from the Pixhawk to the ground control station (GCS). In our case, the GCS is a laptop. The flight controller sends telemetry data via UART connection to the RPi, which transfers the data over WiFi via MavLink Protocol[7].This telemetry data consists of (but not only) battery voltage, drone position, drone attitude, motor input values, and debugging/logging messages. The Vicon Positioning system is directly logged into the ground control station. All information that is sent back to the GCS can also be captured by the Pixhawk on a local micro SD card. The total build can be seen in figure 4.7.

[6] https://fusion.engineering/tech/bidirectional-dshot/
[7] https://ardupilot.org/dev/docs/mavlink-basics.html

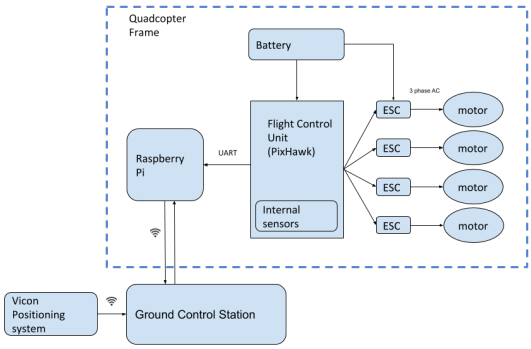


Figure 4.6: Schematic of first build

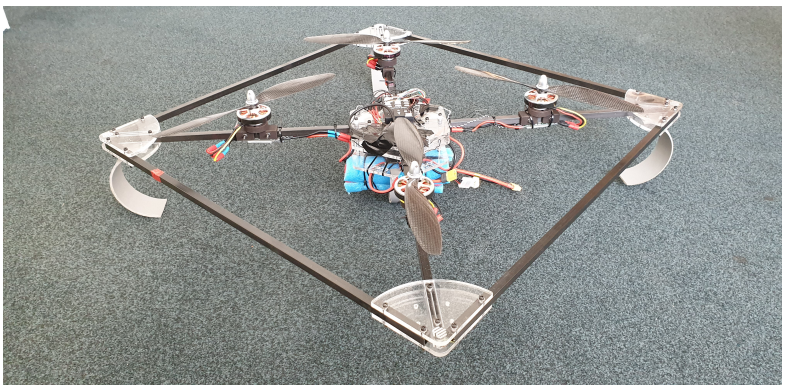


Figure 4.7: Drone build with Pixhawk controller

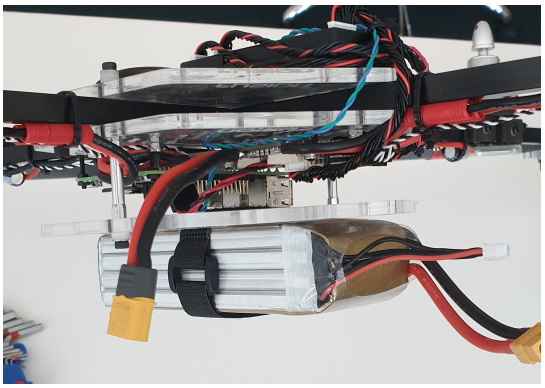


Figure 4.8: Bottom half detail showing the battery and the RPi

4.4.2. Second Build: RPi controlled quad

In the second build, the Pixhawk flight controller is removed from the setup. The purpose of this build is to have Reinforcement Learned agent perform waypoint tracking within the cyberzoo environment. The absence of the internal sensors of the Pixhawk is compensated by using the Vicon Motion capture system. This system can measure quadcopter position with sub-millimeter precision as well as attitude measurements with the same precision. The Vicon system sends the position to the ground control station. The GCS then sends this information via WiFi to the RPi flight controller. The RPi can also send telemetry data back via WiFi, but can also log onto the RPi SD card.

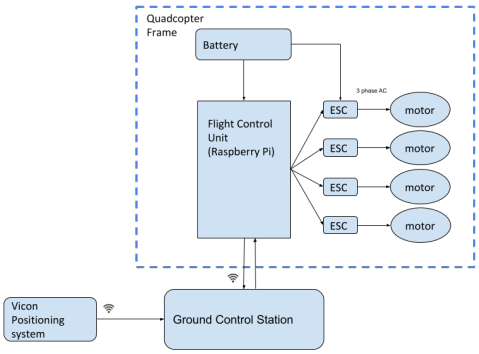


Figure 4.9: Schematic of the second build

4.4.3. Third build: RPi controlled quad in gyro setup

For safety and simplicity, also an attitude control policy is developed. This attitude control policy is tested in the Fusion Engineering gyroscope test setup, see figure 4.11. The gyro setup bounds the position of the quadcopter. Since the policy developed does not contain a separate loop for attitude and position control, a trained policy for position waypoint tracking will not function in the confined test setup of the gyro setup. Therefore a separate attitude control policy is created. The test-gyro setup cannot use the Vicon tracking system for attitude measurements. Instead, each ring of the gyro test setup contains two rotary encoders, capable of measuring up 0.0004 degrees precision. The two encoders of each ring can be averaged to account for non-planar deflections of the drone to non-perfect stiffness of the gyro structure.

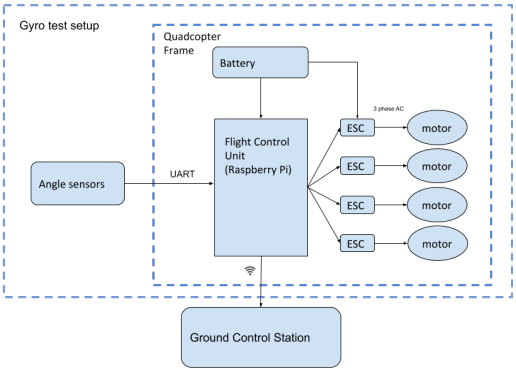


Figure 4.10: Schematic of the second build

Figure 4.11: Fusion Engineering gyro test setup

5

quadcopter System Identification

5.1. Method

The challenge of transferring a policy from simulation to real world is called *sim-to-real* transfer. For the sim-to-real transfer, the model that the agent uses to train should ideally be an identical copy of the real world. If the simulation model and the real world are identical, the behaviour of the quadcopter will be identical to its behaviour in simulation. If the simulation model differs from the real world, the behaviour will also differ from simulation. Different methods exist to solve this problem of transferring a policy from simulation to reality. Often used is training with 'domain randomization', where per episode environment parameters are randomized within a certain range [38] [28]. The advantage of this method is that the agent will view the real life situation as just one other variant of the environment. A disadvantage however is that performance in different compositions of parameters can heavily vary. In the situation of flying the drone the parameters are fixed and experiments can be conducted in supervised environments with steady conditions. Therefore we try to solve the problem of sim-to-real transfer by creating an accurate model by means of parameter estimation. The method consists of two steps:

1. Initial parameter estimation
2. Fine tune parameters using Simulink parameter estimation

First the model structure of the system dynamics is chosen. The model structure will implicitly determine the parameters that need identifying. After initial parameter estimation, these values are fine tuned using the Matlab parameter estimation toolbox. The first step is performed on the real life drone. The second step is done in simulation, to verify if the method could work on the actual drone. The steps to make this work on the real quadcopter are discussed in the 'Future Work' section in chapter 8.

5.2. Two-step identification process

5.2.1. Initial parameter estimation

System identification and parameter estimation is used to create an accurate simulation of the real system. The first step is choosing the model structure after which a list of parameters to estimate is set up. In the case of the quadcopter, some parameters are more easily measured than others. Parameters such as mass can just be determined [40]. Inertia properties are more difficult to measure and have to be estimated [24].
From the dynamics equations in chapter 3 the necessary parameters can be distilled. A few assumptions are made regarding the identification of parameters:

1. Low flight speeds resulting in negligible air drag
2. The quadcopter is symmetric about both x and y axis
3. Parameters don't change during flight

4. CoM and CoG coincide with geometrical centre
5. Mass moment of inertia matrix is diagonal

Because the test flights are performed with low speeds, the air drag is neglected. This is in accordance to Sun's paper on *Identification of quadcopter Aerodynamic Model from High Speed Flight Data*: If flight speeds are below approximately $11m/s$, drag is negligible [36]. The parameters to identify are:

- mass $'m'$ [kg] (equation 3.13)
- distance of motor to axis $'l'$[m] (equation 3.12)
- thrust coëfficient $'c_t'$ (equation 3.12)
- torque coëfficient $'c_\tau'$ (equation 3.12)
- diagonal inertia matrix $'I'$ [kgm^2] (equation 3.13)

Mass

Mass of the system is identified by putting the quadcopter on a scale. The scale used is a portable electronic scale with an accuracy of 5 grams. The measurement of the quadcopter is a mass of $m = 2.545kg$.

Distance motor to axis

The distance of the motors to the x and y axis are measured by ruler and measure $l = 0.170m$.

Thrust and torque coëfficient

Thrust and torque coëfficients cannot be measured directly, but instead need to be estimated. Equation 3.10 describes the relation between the thrust force, the thrust coëfficient and the rotor speeds of a rotor. Rewriting this equation shows that the thrust coëfficient is proportional to the thrust generated divided by the square of the rotor speed (equation 5.1):

$$c_t = \frac{F}{\Omega^2} \tag{5.1}$$

Similarly, the torque coëfficient is proportional to the torque generated divided by the square of the rotor speed (equation 5.2):

$$c_\tau = \frac{\tau}{\Omega^2} \tag{5.2}$$

In order to determine these coëfficients, the thrust force, torque and rotor speeds need to be measured. For this a testbench setup is used. This setup uses two load cells are used: one to measure the force in the thrust direction and one to measure the force perpendicular to the thrust direction, to calculate the torque from. The testbench is depicted in figure 5.1. The testbench holds the motor and load cells in a 3D printed mount, as well as the sensor used for rotor speed measurements. The side of the motor is equipped with 4 reflective tape markers. The rest of the side is covered in non-reflective black tape. The reflective tape reflects the wave from the emitter back to its receiver, while the non-reflective tape does not. Every 4 reflections will count as one revolution. The measurements will take place at a frequency of $50Hz$.

Measurement input:

A PWM signal ranging from 1100 to 2000 is given as input, with steps in between of 1000, and a duration of 1 second per step.

The rotor speeds and forces in both load cells are measured as output.

When plotting the rotor speed on the x-axis and thrust force on the y axis, a clear second order relation is visible between the two. The

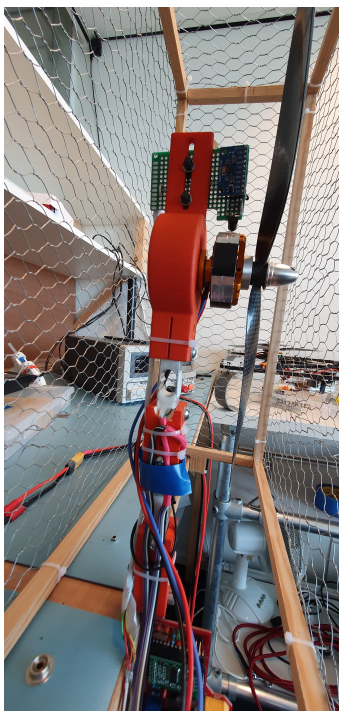


Figure 5.1: Rotor testbench setup

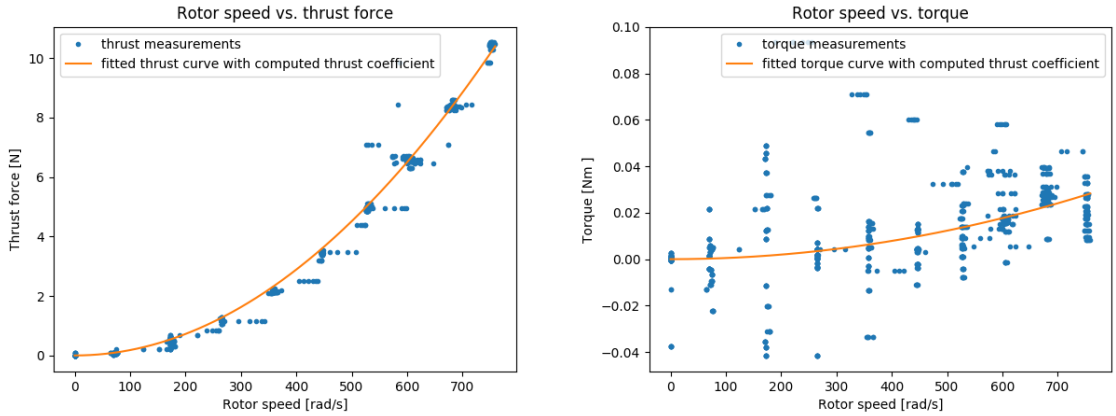


Figure 5.2: Testbench measurements

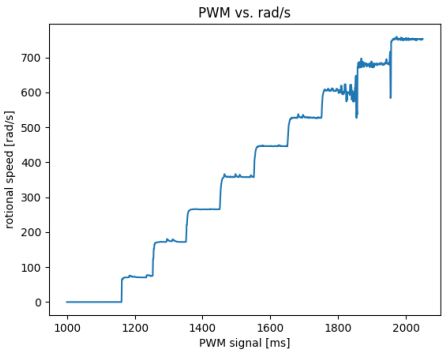


Figure 5.3: PWM signal vs. rotational speed

figures from the rotor speed vs. thrust and rotor speed vs. torque are depicted in figure 5.2. The relation between PWM signal and rotational speed is visible in figure 5.3

A linear regression is used to compute the thrust coëfficient c_t. The error term used for this regression is equation 5.3, where $y(k)$ is the measured output and $\hat{y}(k,\theta)$ is the output of the proposed model, with index k and parameter θ.

$$\varepsilon(k,\theta) = y(k) - \hat{y}(k,\theta) \tag{5.3}$$

Minimizing this error is done using a linear least square method [22].

$$V(k,\theta) = \frac{1}{N}\sum_{k=1}^{N} \varepsilon(k,\theta)^2 \tag{5.4}$$

The resulting model parameters are then

$$\hat{\theta} = min(V) \tag{5.5}$$

Where V is called the cost function, θ is the parameter vector, N is the number of data points, k is the indexand ε is the error vector.

Our proposed model is $\hat{y} = \phi\theta$, where ϕ is the input (vector), y is the output (vector) and θ is a parameter. The error term is then

$$\varepsilon = y - \hat{y} = y - \phi\theta$$

The substituting this value for the error term in the cost function gives

$$V(\theta) = \frac{1}{N}\varepsilon^T\varepsilon = \frac{1}{N}(y - \phi\theta)^T(y - \phi\theta) = \frac{1}{N}(y^t y - 2\theta^T\phi^T y + \theta^T\phi^T\phi\theta)$$

The best model parameters are then solved by minimizing V, as in equation 5.5. The analytical solution is derived by differentiating V with respect to θ

$$\frac{\partial V}{\partial \theta} = 0 \rightarrow \frac{1}{N}(-2\phi^T y + 2\phi^T\phi\theta) = 0 \tag{5.6}$$

Which results in

$$\hat{\theta} = (\phi^T\phi)^{-1}\phi^T y \tag{5.7}$$

Applying the linear least square on the data, a thrust coëfficient and torque coëfficient of $c_t = 1.804e-5[\frac{N}{(rad/s)^2}]$, $c_\tau = 4.907e-8[\frac{Nm}{(rad/s)^2}]$ are found.

Moment of Inertia

Sun's paper on *Identification of Quadrotor Aerodynamic model from High Speed Flight Data*[36] uses a method proposed by Mendes et al. in the paper *Determining moments of inertia of small UAVs: A comparative analysis of an experimental method versus theoretical approaches*[24], which uses the masses and simple inertia estimates of the individual components to compute a total inertia matrix of the system. Their method supposedly has an accuracy of 5%. Their method is preferred over other methods, since it requires no additional test setups.

The centre pad is modelled as a flat cylinder, with radius $r = 0.085m$, length $l = 0.10m$ and mass $M = 1.545kg$. The motor assemblies that include the motor mount, motor, rotor, ESC and the bolts and nuts are modelled as a point mass of $m1 = 0.130kg$. The corner pieces that include the plexi-glass corners, the legs and the bolts and nuts are also modelled as a point mass of $m2 = 0.120kg$. In total this results in one centre pad with cylindrical shape and eight point masses. The model is depicted in figure 5.4

Figure 5.4: Inertia model with cylindrical mass M in the centre and point masses m1 and m2 as motor assembly and corner assembly

The inertia around x and y axis of the cylinder (where the inertia around x and y axes are identical) and inertia around the z axis are calculated with equation 5.8.

$$I_x = \frac{MR^2}{4} + \frac{Ml^2}{12} \tag{5.8a}$$

$$I_z = \frac{MR^2}{2} \tag{5.8b}$$

For the point masses, the inertia around the respective axis is a function of their mass and the square of the distance, see equation 5.9.

$$I = mR^2 \tag{5.9}$$

The total inertia around x, y and z axes are the sum of each of the components: the cylinder, the four inner point masses of the motor assembly and the four outer point masses of the corner pieces, see equation 5.10.

$$I_{xtot} = Ix_{cyl} + 4I_{xmotor} + 4I_{xcorner} \tag{5.10a}$$

$$I_{ytot} = Iy_{cyl} + 4I_{ymotor} + 4I_{ycorner} \tag{5.10b}$$

$$I_{ztot} = Iz_{cyl} + 4I_{zmotor} + 4I_{zcorner} \tag{5.10c}$$

Using these equations, the resulting inertia numbers are depicted in table 5.1.

	I_{x}	I_{y}	I_{z}
Center cylinder	0.004078	0.004078	0.005581
Inner point mass (motor assembly)	0.003757	0.003757	0.007414
Outer point mass (corner assembly)	0.008748	0.008748	0.01745

Table 5.1: Table of inertia values for the individual components

Plugging these values in the equations 5.10 give total inertia values of:
$I_{xtot} = 0.054098 \quad kgm^2$
$I_{ytot} = 0.054098 \quad kgm^2$
$I_{ztot} = 0.104037 \quad kgm^2$

The gyro test setup also adds inertia to the system. The inetia of these parts is taken from the analysis of the 3D CAD files provided by the designer.

5.2.2. Tune the parameters

This step of is tested in simulation only. Read the section on future work 8.2 for the steps needed to implement this on the real quadcopter.

To fine-tune the parameters found in the previous section, a parameter optimization is developed in simulation. The Matlab toolbox *parameter estimation* will be used in combination with Simulink. The dynamics model of equation 3.13 is implement into a Simulink block, as well as the controller proposed in section 3.5. The Matlab parameter estimation toolbox compares the output data of the Simulink model y_{sim} to reference data y_{ref}. The estimation error is formulated as:

$$e(t) = y_{ref}(t) - y_{sim}(t) \tag{5.11}$$

The parameter optimization minimizes this error function by minizing the cost functions, which in this case is the sum of the squared error:

$$F(x) = \sum_{t0}^{tn} e^2(t) \tag{5.12}$$

Because no real flight data is yet used, simulation is used for generating data as well. The same Simulink model is used, but with different initial parameters. This model will be called the *data generation model*. The model that needs identifying of parameters will be called the *transition model*. This name is chosen because the transition model term is also used in Reinforcement Learning as the dynamics model that transitions the environment from one state to the next after an action is taken. A *trajectory generator* generates position set-points. The controller takes in these set-points and the current position of the quadcopter to generate inputs for the quad. These inputs enter both the data

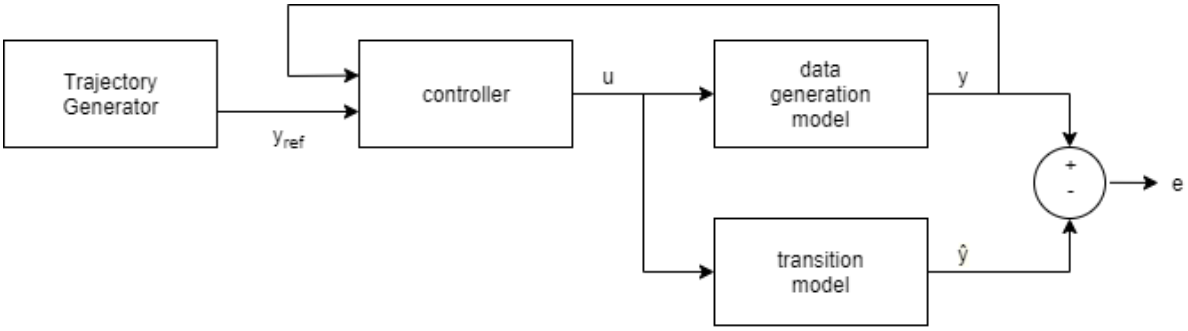


Figure 5.5: Diagram of the parameter optimization finetune method

generation model and the transition model. The difference in the output is the error term used for parameter optimization. This method is captured in the block diagram of figure 5.5.

Trajectory
For estimation a 3D step input is chosen. The 3D step input ensures that all important dynamics of the quadcopter are excited. The starting position is $p_0 = (0, 0, 0)$. The target position is $p_1 = (1, 1, 1)$.

5.3. Test in simulation

The data generation model is responsible for generating the motor inputs u, that will be used as input for the transition model in order to estimate the parameters. The controller in the block diagram 5.5 is identical to the controller discussed in section 3.5.The PD values of this controller consist of position gain K_{p1}, position damping K_{d1}, attitude gain K_{p2}, attitude damping K_{d2} with values:

$$K_{p1} = [1.5 \quad 1.5 \quad 1.5]$$
$$K_{d1} = [2 \quad 2 \quad 2]$$
$$K_{p1} = [160 \quad 160 \quad 160]$$
$$K_{d1} = [40 \quad 40 \quad 4]$$

The parameter values used to generate the data correspond with the initial parameters discussed in section 5.2.1:

$mass[kg]$	2.545
$armlength[m]$	0.17
$c_t[\frac{N}{(rad/2)^2}]$	1.804e-5
$c_\tau[\frac{Nm}{(rad/2)^2}]$	4.907e-8
I_x	0.054098
I_y	0.054098
I_z	0.104037

Table 5.2: Initial estimation of parameters

The parameters mass, arm length, thrust and torque coëfficient are measurable with good accuracy. The inertia however is more difficult to measure and has to be calculated. Mendes et. al [24] claims that a multibody approach is able to yield errors of lower than 5%. To allow for higher faults in inertia estimations, we will test if the parameter estimation technique will be able to estimate the right inertia values, when starting from inertia values multiplied by a factor 10:

$$I_{x2} = 10I_x = 0.540980$$
$$I_{y2} = 10I_y = 0.540980$$

Where I_{x2} and I_{y2} are the inertia values of the *transition model*.

The parameter estimation results after 11 iterations in values of:

$$I_{x2} = 0.05371750$$

$$I_{y2} = 0.05375963$$

Which is close to the original value.

Figures 5.6 to 5.8 show the x,y,z position of three lines: the test data, generated by the *data generation model*, the output of the simulated model *before parameter estimation* and the output of the simulated model *after parameter estimation*. The figures show that after estimation, the simulation data closely resembles the original data and thus estimation of inertia is possible using this method. The plots of the other outputs: u,v,w velocities, roll, pitch, yaw attitude p,q,r rotational velocities can be found in appendix C.

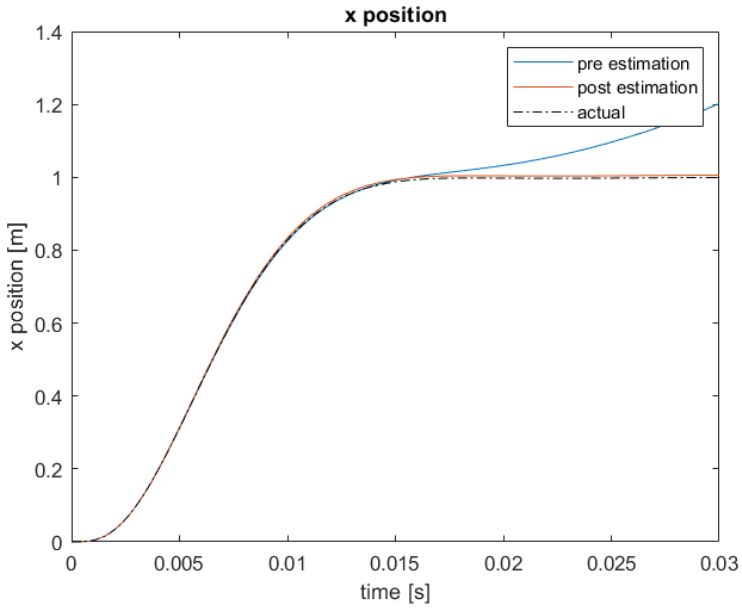


Figure 5.6: x position of actual data, the simulated output pre-estimation, the simulated output post-estimation

5.4. Conclusion

This chapter shows the two-step parameter estimation process of a quadcopter. The first step measures mass, length of arms, thrust and torque coëfficients. The mass and length of arms were measurable easily. For the two coëfficients, a testbench setup was used. The testbench measures rotational speed of the rotor, as well as thrust force and torque. From the resulting data the coëfficients can be estimated. The inertia values are first estimated using a composite model of a centre cylindrical disk and multiple point masses. To test if a mis-estimation of inertia can be corrected in the second estimation step, the inertia's I_x and I_y are multiplied with a factor 10. The Simulink parameter estimation then shows to be able to correct the parameters back to their original values.

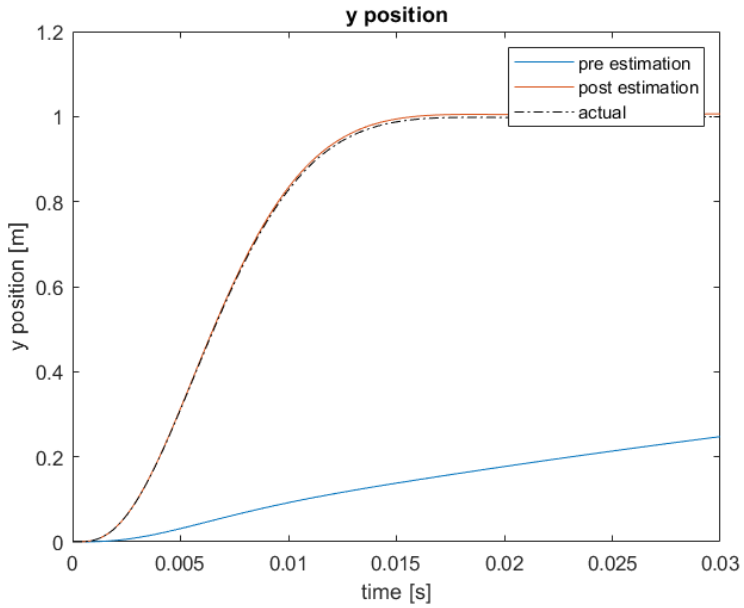


Figure 5.7: y position of actual data, the simulated output pre-estimation, the simulated output post-estimation

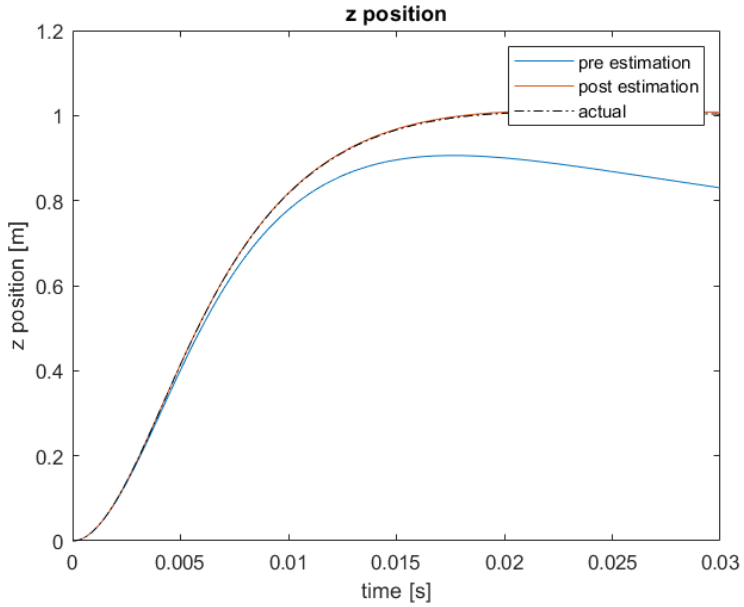


Figure 5.8: z position of actual data, the simulated output pre-estimation, the simulated output post-estimation

6

Implementation of the Reinforcement Learning Based Controller

This chapter will focus on the design and implementation of a Reinforcement Learning algorithm in order to make a quadcopter fly autonomously. A simulation will be created as training environment. The RL algorithm is chosen and the Neural Network architecture and hyperparameters are set, after which training starts. After training, the simulation results will be analyzed.

The target of this thesis is to develop an RL agent that directly maps observations to low level motor commands. This means that the generated policy should take in the states (observations) and output *four motor values*, see figure 6.1.
The algorithm used will be TD3, as discussed in chapter 2 [9].

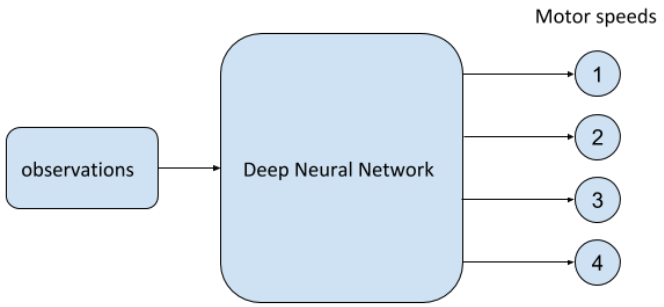


Figure 6.1: General overview of the target goal: Direct mapping from observations to low level motor commands

6.1. Simulation states and boundaries

The simulation environment takes in the current states and actions, and returns the next states and the reward function. It also should keep track of the scores and whether or not an episode is terminated. The dynamics model of chapter 3 is used, with the identified parameters from chapter 4.
The states (or observations) are defined as in chapter 3, with some addition. To be able to know the desired position, the difference between x,y,z and the target x,y,z coördinates is added to the state. The total state vector therefore is:

$$s = \begin{bmatrix} x & y & z & u & v & w & \phi & \theta & \psi & p & q & r & \Delta x & \Delta y & \Delta z \end{bmatrix} \tag{6.1}$$

Where states $x, y, z, \phi, \theta, \psi, \Delta x, \Delta y, \Delta z$ are in inertia frame, and states u, v, w, p, q, r are in body fixed frame. The desired attitude is always assumed to be zero for all three angles and therefore the error

term of the angles is equal to the angle itself.
The states themselves are not bounded. However, a termination statement marking the end of an episode does indirectly bound the states: The termination of an episode is defined if:

- $\phi > \pm\frac{\pi}{2}$
- $\theta > \pm\frac{\pi}{2}$
- $\psi > \pm\frac{\pi}{2}$
- x or $y > \pm 3$ [m]
- $z < 0$ or $z > 3$ [m]

The four motor outputs are bounded to the actual output range of the motors. From the experimental data of chapter 4 the maximum and minimum output of the motors can be determined. The maximum rotational speed of the motors is $\Omega_{max} = 750 rad/s$, while the minimum rotational speed is $\Omega_{min} = 100 rad/s$

6.2. Neural Network architecture

TD3 uses in total six deep neural networks: An actor and a target actor network. The actor network is also known as the *policy*. This is what will make the decisions. Two critic networks are being used, which both have their own target critic network. In accordance to Fujimoto et al. [9], the neural network structure of all six neural nets has one input layer (15 nodes), two hidden layers (400 and 300 nodes) and one output layer, which is depicted in a schematic in figure 6.2. The critic networks have one output node in the last layer. The actor networks have four outputs in the last layer. Both hidden layers have a ReLu activation layer.

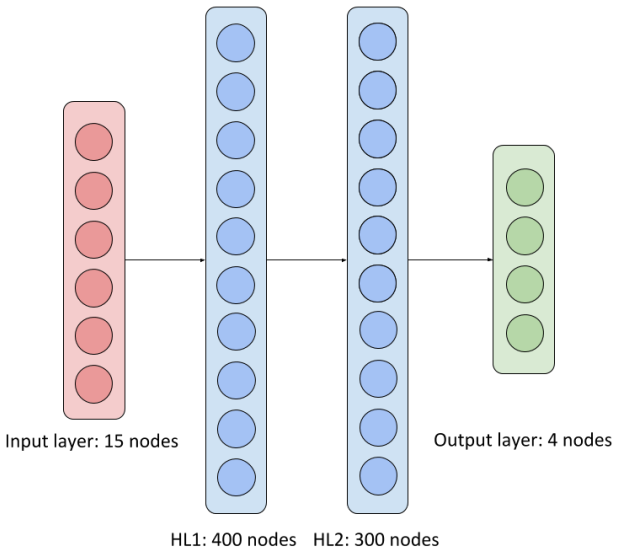


Figure 6.2: Schematic of the neural network architecture for the actor and target actor network

Input layer (15 nodes) –> Hidden layer 1 (400 nodes) –> ReLu activation layer –> Hidden layer 2 (300 nodes) –> ReLu activation layer –> Output layer (4 nodes for the actor(s), 1 for the critic(s)).

The output of the actions should be bounded between -1 and 1. These values are scaled to the real values of rotational speed of the motors so that the -1 bound corresponds to $\Omega_{min} = 100$ and the +1 bound corresponds to $\Omega_{max} = 750 rad/s$. All speeds in between are linearly interpolated.

6.3. TD3 hyperparameters

The learning rate is set at $\alpha = 7e-4$ for both the critic and actor networks. The original paper of Fujimoto et al. uses a learning rate of $7e-3$ for the critic and $7e-4$ for the actor networks, but hyperparameter tuning showed that a smaller learning rate for the critic networks resulted in better performance. The discount factor γ is set to be 0.99. The noise $\mathcal{N}(0,\sigma)$ that is added to the actions for exploration purposes has zero mean and a variance of $\sigma = 0.2$. The noise is clipped to a maximum of half the action range so that the target update y equals:

$$y = r + \gamma Q_{\theta'}(s', \pi(s') + \epsilon, \epsilon = clip(\mathcal{N}(0,\sigma), -c, c) \tag{6.2}$$

Where $c = 0.5$.

A batch size of 256 is used, with a Replay Buffer size of $1e6$. This size can be increased if computational capacity of the user computer allows it. Larger batch sizes tend to decrease performance, as measured by the ability to generalize. This is due to sharp minimizers in the test and training functions. Smaller batch sizes tend to have flatter minimizers, which is due to noise in the gradient estimation [18]. The maximum time of an episode is 500 time steps. With a project step time of $t_s = 0.02s$, or 50 Hz, results in a maximum episode time of $t_{max} = 10s$.

6.4. Simulation target, initial conditions and reward functions (task specific)

In RL, two major types of reward functions are possible: Sparse reward functions and dense reward functions. Sparse reward functions are rewards only given in certain states and are generally high in nature. In the case of the lunar lander, a small negative reward is given on actions, and a high reward of 200 is given for landing within the target. Landing outside the target yields a reward of 100. Crashing the lander yields a reward of -100. These rewards/penalties are have a large value but are given sparsely. Sparse rewards focus on the *outcome* of an episode. Dense reward functions give smaller rewards more often. This can be a reward for smaller sub-tasks, or a reward depending on a distance to a point. This last version is how we will implement the reward function. The advantage of sparse rewards is that they are relatively free of choice: No information in how the reward is reached is given. However, in a situation where rewards are to scarce, the agent will never reach a reward 'by coincidence'. Also it might be difficult to define good behaviour. In the case of position tracking, what is defined as a good position? The risk of using a dense reward function however, is that the designer of the function imposes too much of knowledge of how to solve a problem into the system. The way to the solution should be free to be developed by the agent, not superimposed. The reward function for all tasks has the same form, but different parameters:

$$r = \sum \left(\tanh\left[1 - p_1\Delta(x,y,z)^2 - p_2\Delta(\phi,\theta,\psi)^2\right]\right) - p_3\frac{\partial u}{\partial t}^2 - p_4(u - u_{base}^2) \tag{6.3}$$

For stability in training, a positive definite function is desired. Therefore, the reward function is multiplied with a ReLu function. This function is also as activation layer in neural network and makes sure the output is positive definite:

$$f(x) = \max(0, x) \tag{6.4}$$

$$r = \begin{cases} \sum \left(\tanh\left[1 - p_1\Delta(x,y,z)^2 - p_2\Delta(\phi,\theta,\psi)^2\right]\right) - p_3\frac{\partial u}{\partial t}^2 - p_4(u - u_{base}^2) : r > 0 \\ 0 : r < 0 \end{cases} \tag{6.5}$$

Where p_1, p_2, p_3, p_4 are the parameters that define the weight of each term. The two delta terms in the equation are the difference between target x and x, target y and y, target z and z, and target angles. It results in three hyperbolic tangent values that will be added together. This value will diminished by the derivative of the action with respect to the time squared. This method is described by Martinsen [23] to reduce oscillations in the actions. The parameter corresponding with the derivative of the action is usually small. The last term is the difference between the input and a set base value. In normal 3D flight, this base value is irrelevant because the policy will adjust it's rotor speeds to increase or reduce throttle. However in hover, only a difference in rotor sets is of importance. The hover tests

will be performed in an indoor environment. To reduce noise levels and wind gusts produced by the quadcopter, the base level is chosen conservatively with respect to normal rotational speeds for hover.

6.4.1. Hover

The hover policy is trained so that the resulting policy can be used in the gyro test setup. This means that the position of the quadcopter is stationary. The states of the quadcopter are hardcoded to always have the same position and the target position is hardcoded to coincide with the current position. No position changes happen, so the linear velocities are also zero.
Target pose is an angle of 0 degrees for all angles. The altered state vector is:

$$s = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 & 0 & \phi & \theta & \psi & p & q & r & 0 & 0 & 0 \end{bmatrix} \tag{6.6}$$

The initial attitude is randomized per episode. Roll, pitch and yaw angles are randomly chosen between -80 and 80 degrees. The randomization of the initial angle increases generalization of the policy.
The hover policy is trained in the gyro test setup. This test setup consists of multiple rings with large diameter. The inertia of the rings is added to the inertia of the quadcopter in simulation, so that the sim to real transfer stays accurate.
Since we are not interested in any position error, the parameter corresponding to the position error is 0: $p_1 = [0, 0, 0]$. The angular error is the important factor in this case so parameters corresponding to the angular error is $p_2 = [20, 20, 10]$, where the order of angles is ϕ, θ, ψ. The negative reward on change in rotor speeds is $p_3 = 0.8$. The base level for hover is chosen to be at half of hover thrust (based on the mass of the quadcopter and the thrust coefficient of the motors). Corresponding parameter is chosen at $p_4 = 0.03$. Concluded, the four parameters for the reward function are:
$p_1 = [0, 0, 0], p_2 = [20, 20, 10], p_3 = 0.8, p_4 = 0.03$

6.4.2. Position tracking

The position tracking policy's target is to always reach the point $(x_{target}, y_{target}, z_{target})$. To keep this policy generalizable from any starting point, the initial x,y position of the quadcopter varies between (-1,1). The initial z position is -0.02. To assure that the quadcopter can also take of from non-ideal flat surfaces, the initial attitude varies between angles from (-8.0,8.0) degrees, for all three ϕ, θ, ψ angles. The target position is changed every *500 time steps*. This way the agent learns to track different positions. The target positions are randomized values between (-1,1) for x and y position and (-2,-1) for z position. In contrast to the reward function parameters for hover, now also the position error should play a role in the policy.
Two different types of reward parameters are implemented:

1. negative reward on both attitude and position error
2. negative reward on position error only

The first type still penalizes errors of the attitude by the same amount as in the case of hover, so $p_2 = [20, 20, 10]$. It also has an added negative reward for position error, parameter $p_1 = [3, 3, 3]$. The parameter for position error is chosen by trial and error. Choosing values that are too high results in unstable training: The policy will not converge to anywhere close to optimal performance. Choosing values that are too low make the agent ignore the position and just focuses on the attitude error of the quadcopter.

The second type does not involve a negative reward for attitude error. The parameter corresponding to the attitude is zero: $p_2 = [0, 0, 0]$. This reward function does have a high negative reward on position control: $p_1 = [10, 10, 20]$. This reward function imposes less restriction on the policy, since all knowledge about stabilizing the attitude is left out.

The small negative reward for changes in rotational speed seemed to be working with the hover case, so it is kept at $p_3 = 0.8$. Experimentation with this parameter also revealed that increasing this value to high results in a 'lazy' policy: The agent would rather have a stable action output than changing its attitude or position. The final parameter corresponding to the base level is not needed in the non-hover case, so $p_4 = 0$.

7

Results

This chapter training and testing results of the simulation of both the hover and the position policies. Training is performed on an HP z-book studio g5, with an intel i7 CPU. Training the neural networks is done with the use of NVIDIA CUDA, on an Nvidia Quadro P-1000 GPU. Implementation is done using the python package Torch. Torch has a build in CUDA implementation, where selecting the device is relatively straight-forward.
We train for two different tasks: The first task is hover. The position of the quad is fixed. The second task is position control (or way-point tracking). For this task we train two policies, each with their own type of reward: *Type 1* has a negative reward for both position and attitude error. *Type 2* only has a negative reward for position error. The reward structure, hyperparameters and algorithm are implemented as explained in chapter 6. The simulation uses a global time-step of $0.02[s]$.

7.1. Hover training

As visible from the this figure, it took quite some episodes before the agents learns to not flip: only after around 2000 episodes, the quad is improving stability.

Figure 7.1: Hover training result: reward per episode

On the x-axis the number of episodes is plotted. On the y-axis the reward for that episode is plotted. The blue dots in the graph are all the measured scores for each episode. The red line is the average reward over hundred episodes. When looking at the red line, the agent shows clear signs of learning behaviour. From figure 7.1, it seems that it takes a long time before the agent starts learning. However, looking at the same data differently, a different insight can be gained: Figure 7.2 shows the rewards not per episode, but per total amount of time steps. The time steps of the early episodes are very limited because of episode termination. When improving the policy, the episodes will last long (while still learning). Plotting reward vs. total time steps shows a steady increase of reward per time step.

Figure 7.2: Hover training result: reward per time step

7.2. Hover testing

While training the policy, the TD3 method requires exploration by adding noise to the actions. In testing however, that noise is unwanted. For testing, the noise is removed, but the target attitude and initial conditions remain the same. In figure 7.3 an overview of testing results are visible. The testing length is 1200 time steps. The test is a collection of 100 episodes. The lightest color in plots is the area between the maximum and the minimum position per time step. This shows an area of all possible trajectories, when starting from randomized initial conditions. The area with the darker color shows the standard deviation of each time step: This depicts a spread of all 100 episodes. The dark line is the plot of one of the episodes, to show what an individual trajectory of angles would look like.

Figure 7.3 shows converging behaviour for all angles. To define converging/stabilizing, a metric from conventional control theory is used: *rise-time*. Stabilizing is defined as achieving a roll and pitch angle of under *0.05 %*. Figure 7.4 shows a close up of the first five seconds of testings. The rise time boundaries are shown as the black dotted lines in pitch and roll plots. The max-min spread at the starting point is equal to the spread of the initial angles (as expected). Since initial conditions are chosen at random between -1 and 1, the standard deviation spread is seen to be coinciding with the 68% confidence interval (also as expected). Convergence of the minimum and maximum values of the angles show a stable policy, always converging to zero.

Looking at the standard deviation plot, the rise time is *1.58 [s]* seconds.

Not all episodes result in successful behaviour. An episode is successful if the maximum amount of time steps is reached without terminating the episode ($t = t_{max}$). When an episodes terminates prematurely, this is counted as a fail. The success rate of 100 episodes is 94%.

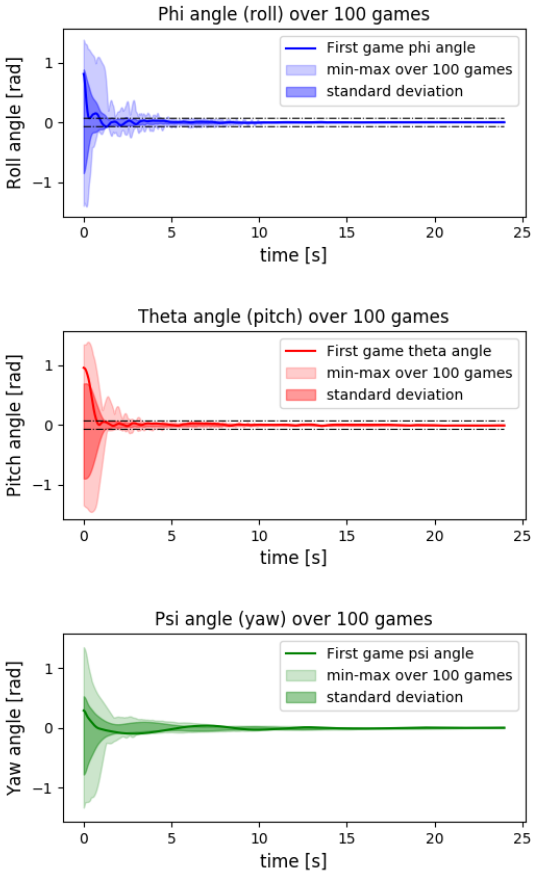


Figure 7.3: Hover test

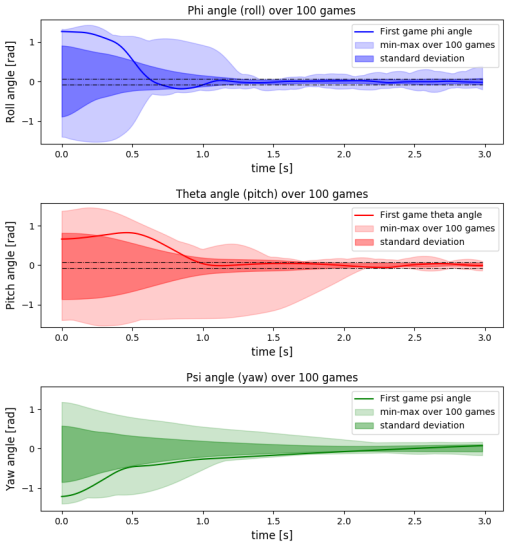


Figure 7.4: Hover test (close up)

7.3. Position tracking training

Figure 7.5 and figure 7.6 show the reward per episode and the reward per time steps of the *type 1* reward function: negative reward on both attitude and position error, as well as the *type 2* reward function: negative reward on only position error. The dots of the scatter plot in both figures represent the data points. The dark lines represent the average of the last 100 data points.
For *type 1* (the orange line) the agent improves over time with a peak at 7500 episodes (approximately 2.9e6 time-steps), with a maximum average score of *7580*. However, after the peak, performance declines fast. The performance drop might be causes by discontinuities in the Q function. As was also the case for training the hover policy, the agent seems not to learn anything for the first 6800 episodes. However, the maximum time for an episode in case of the position tracking agent is *5000 time steps*. When plotting reward vs. time steps, it is clear that the agent does indeed learn almost from the start. The *type 2* (the red line) reward function results in a working policy in around 1200 episodes (approximately 1.1e6 time steps), with a maximum average score of *9100*. Some decrease is then visible, though is less extreme as is the case with the *type 1* reward function. Penalizing only the position results in quicker learning curve, a higher average score and more stable learning behaviour. *Remark: The higher average score is less relevant, since a different reward function will yield different values of reward for same performance.*

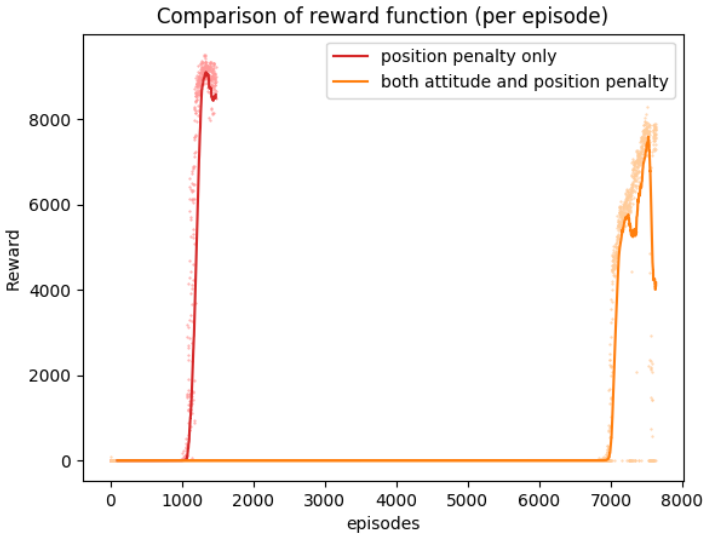


Figure 7.5: Comparison of two reward functions: type 1 (orange) and type 2 (red)

Using the *type 2* reward function in combination with the DDPG algorithm shows that this algorithm is not capable of learning way-point tracking behaviour. After a small amount of time-steps, the agent learns how to get a reward of 30. However it does not learn anymore after that. It's behaviour results in the same rewards for the next $4e5$ time-steps, see figures 7.7 and 7.8.

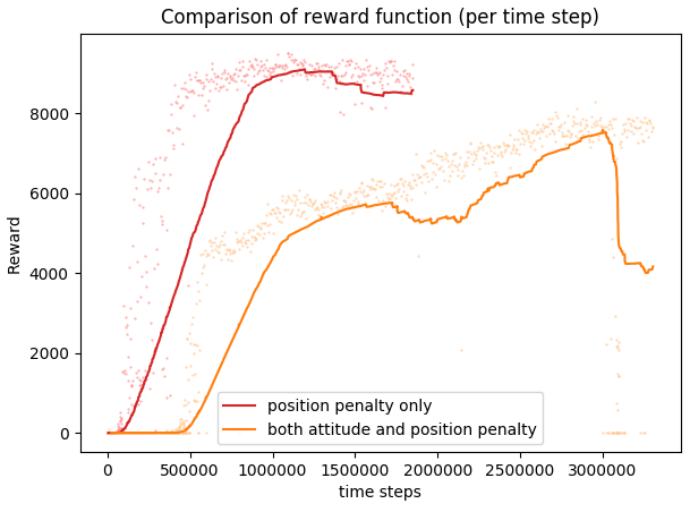


Figure 7.6: Comparison of two reward functions: type 1 (orange) and type 2 (red)

Figure 7.7: DDPG algorithm with type 1 reward function, reward per episode

7.4. Position tracking testing

The Actors neural network values from the peak performance of both reward types *type 1: attitude and position error negative reward* and *type 2: position error negative reward* are used to perform way-point tracking. Four different trajectories are used to analyse flight behaviour:

1. liftoff
2. z
3. square
4. square+z

The first test performed is a z-varying trajectory called trajectory **liftoff**: The quadcopter starts at the origin. This trajectory has only one target way-point

Figure 7.8: DDPG algorithm with type 1 reward function, reward per episode

$wp_1 = (0, 0, -1)$.

The second trajectory is called **z**. The target x and y location remain at coördinates (0,0). Target Z position increments with *-0.5 m* every 1000 time steps to generate six way-points:

$wp_1 = (0, 0, 0), wp_2 = (0, 0, -0.5), wp_3 = (0, 0, -1), wp_4 = (0, 0, -1.5), wp_5 = (0, 0, -2), wp_6 = (0, 0, -2.5)$.

The third trajectory is called **square**. The target position changes every 1000 time steps to follow the form of a square with five way-points:

$wp_1 = (0.8, 0.8, -1.5), wp_2 = (-0.8, 0.8, -1.5), wp_3 = (-0.8, -0.8, -1.5), wp_4 = (0.8, -0.8, -1.5), wp_5 = (0.8, 0.8, -1.5)$.

To also test the ability to fly in a square pattern with different altitudes, the trajectory **square+z** is created: The target position changes again each 1000 time steps, with same x and y positions as trajectory square, but with different z positions.

$wp_1 = (0.8, 0.8, -1.8), wp_2 = (-0.8, 0.8, -1.5), wp_3 = (-0.8, -0.8, -0.4), wp_4 = (0.8, -0.8, -0.8), wp_5 = (0.8, 0.8, -1)$.

The trajectories for both reward types can be seen in figures 7.9 to 7.12. All gathered measurements including isolated x,y,z plots, motor commands, attitude values and position error plots can be found in appendix A. Both policies track the way-points. However, as can be seen by the trajectory figures, the *type 1* policy never reaches a steady state. Instead it keeps oscillating around the way-point, no matter where the way-point is in three-dimensional space. The motor commands keep fluctuating between maximum and minimum motor value. This makes the policy highly unusable for real life implementation. The *type 2* policy however, reaches a steady state more often, even if there is a steady state offset. The offset has a maximum of 0.1 m. The oscillation vs. steady state behaviour can be observed also when looking at the isolated x,y,z behaviour and the motor commands in appendix A.

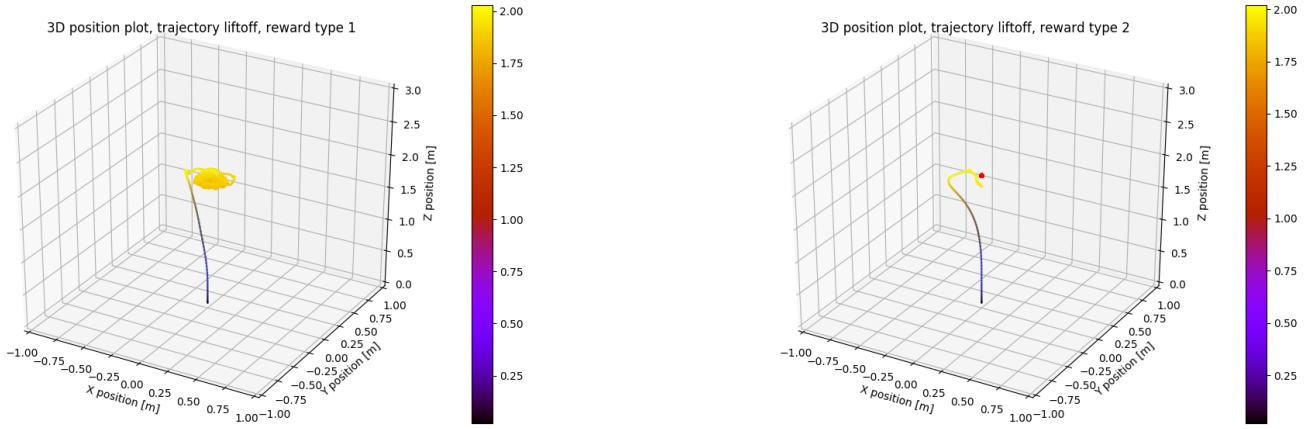


Figure 7.9: Liftoff trajectory

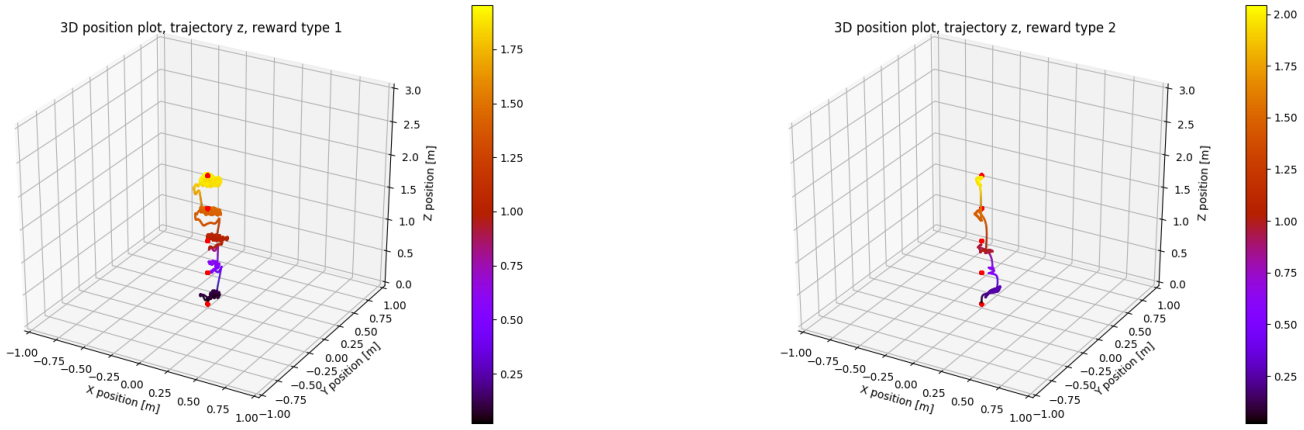


Figure 7.10: z trajectory

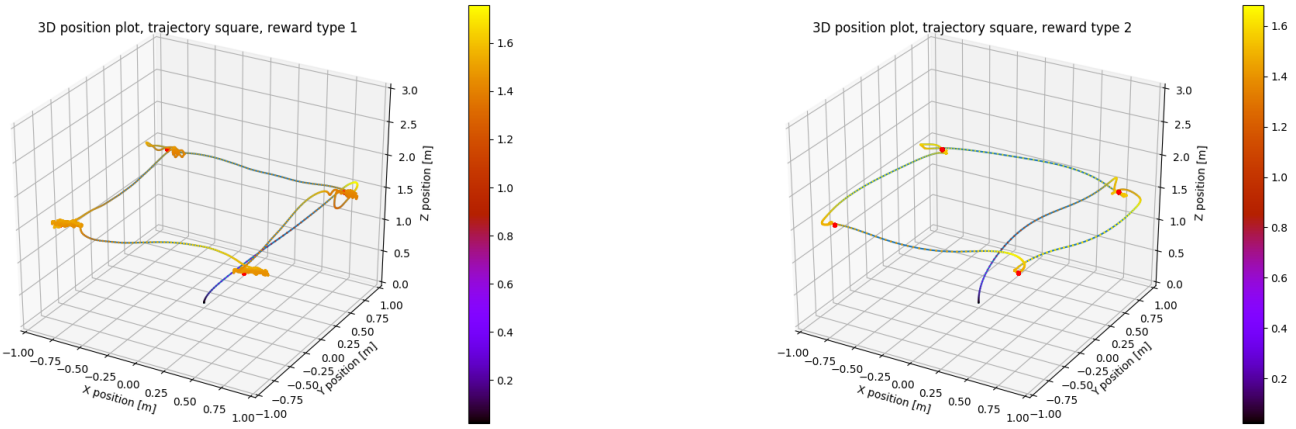


Figure 7.11: square trajectory

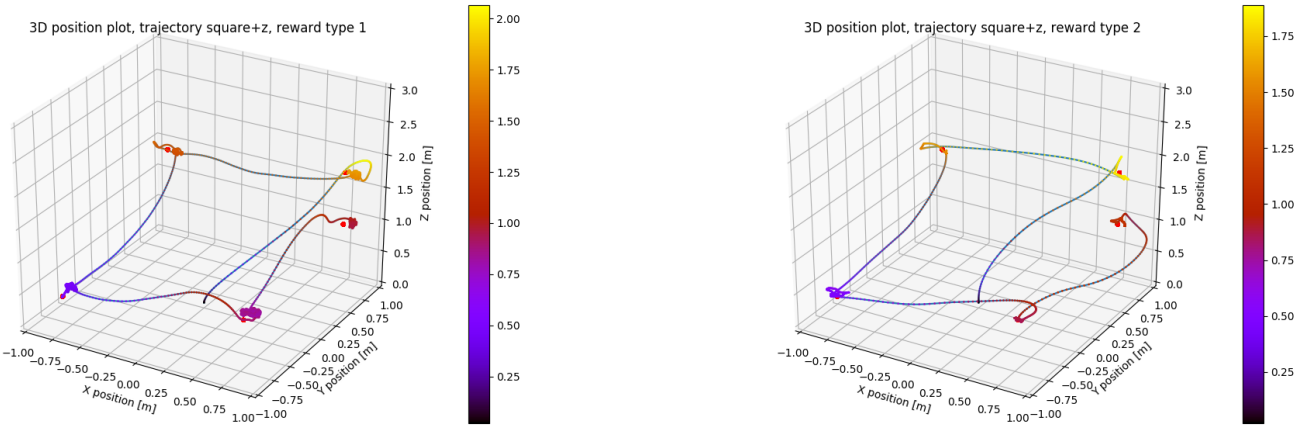


Figure 7.12: square+z trajectory

8

Conclusion

8.1. Conclusion

Reinforcement Learning has great potential in the field of games as well as the field of robotics. Google's DeepMind shows the applicability towards games and Hwbango et al. [13] show the applicability on real life robot platforms. This thesis investigates the possibility of achieving full low-level quadcopter control using RL. The goal was to replace both inner and outer control loop, so that the policy computes motor commands directly from sensor input.

The thesis investigates different Reinforcement Learning algorithms, and finds an RL algorithm suited for this task. The action space of the problem is continuous, since all four motor commands are continuous. Where Q-Learning is not capable of handling continuous action spaces, DDPG and its variant algorithm TD3 are capable of handling these action spaces. TD3 is a more advanced version of DDPG with three extra features with respect to DDPG:

1. Clipped double Q-Learning
2. Delayed policy updates
3. Target policy smoothing

The first trick prevents over estimations in the Q-function. The second trick helps stability in training. The third trick makes exploitation of errors in the Q-function harder by adding noise to the actions.

The structure of the reward function is chosen to be dense of nature rather than sparse. The continuous nature of the tasks at hand (way-point tracking) makes this option preferable. Also, when training for tasks for which the target states are not likely to happen 'by accident', sparse rewards are not preferable. For the hover task, the reward function penalizes the attitude error heavily, and adds a small negative reward for the time derivative of the motor commands. For the position control task, two types of reward function are implemented: *type 1* penalizes both attitude and position error, while *type 2* only penalizes position error.

The hover task is successfully accomplished, with in total 3500 training episodes (around 6e5 time steps). Training is stable and performance does not decrease after longer amounts of time. Testing the hover capabilities is done by initializing a random attitude between -80 and 80 degrees for all three angles roll, pitch and yaw. The metric for stabilizing is lend from classic control theory and is in the form of 'rise time'. Rise time is 1.58 seconds, with a success rate of 94%.

The two types of reward function implemented for position control both generate a working policy. *Type 1* (both attitude and position error negative reward) yields a working policy after a training period of 7500 (around 3e6 time steps) episodes with an average reward over 100 episodes of 7580. After reaching the peak, performance declines rapidly. This is most likely the result of errors in function approximation, and might be solvable by hyperparameter tuning, or changing the reward function. The

resulting policy has high fluctuations in motor commands and therefore oscillating attitude and position values. In none of the trajectories a steady state value is reached.
The *type 2* (only position error negative reward) yields a working policy after a shorter training time of 1200 episodes (1.1e6 time steps). For all trajectories, this policy achieves steady state for almost each way-point. Steady state error occurs with a maximum deviation of 0.1 m. The steady state error might be solved by incorporating an integrator term into the position error in the state.

To bridge the sim-to-real gap, a framework is developed to create an accurate quadcopter model. The framework consists of a two-step parameter estimation: The first step is measuring the measurable parameters such as mass, geometry, thrust and torque coëfficients and estimating the inertia using the summation of a combination of point masses and a central cylindrical disk. This method has proven to be able to converge inertia parameters with a deviation of a factor 10 in simulation.

This thesis proves that TD3 can be used for low-level quadcopter control, replacing both inner and outer loops of the quadcopter control. Using the dense reward function penalizing position control only, results in a stable policy that can track way-points all throughout the 3D space.

8.2. Future Work

This thesis leaves two main parts to be researched in future works. The first part of future work is implementing the hover and position control policies on the real life quadcopter. The hover control policy can be tested in the gyro test setup, as described in chapter 4. The position control policy should be tested in the cyberzoo test setup of the TU Delft, also described in chapter 4. The states of the policy are chosen to be measurable in the cyberzoo test setup.

The second is the implementation of the two-step parameter estimation on real flight data. In this thesis, the second step of the two-step parameter estimation has been performed by generating flight data in simulation. In future work, flight data can be generated by flying a real life physical quadcopter. A copy of the flight controller used to fly the real quadcopter needs to be recreated in simulation, to facilitate the same actuator inputs in simulation with respect to the real flight controller. The output of the real flight data and the simulated flight data can be compared for the second step of the two-step parameter estimation.

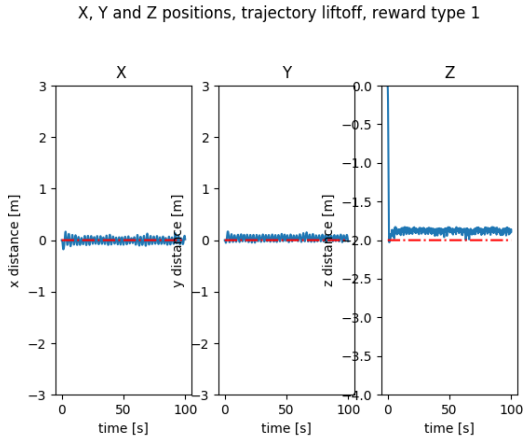
Implementing the proposed framework and the use of TD3 algorithm might be basis for further experimentation regarding other tasks for quadcopters. The tasks now trained for are also achievable using regular control theory. However, by proving this learning framework is effective in applying RL to a quadcopter platform, doors to more difficult tasks have opened: By adding sensors for obstacle detection, an agent might easily learn to avoid obstacles during way-point tracking. Also highly non-linear acrobatic manoeuvres could be learned using this learning framework. Multi agent coöperation and object manipulation can now be experimented with.

By proving low-level motor control is possible, a little piece of the puzzle of 'self learning drones' is solved.

A

Measurements

A.1. XYZ position plots



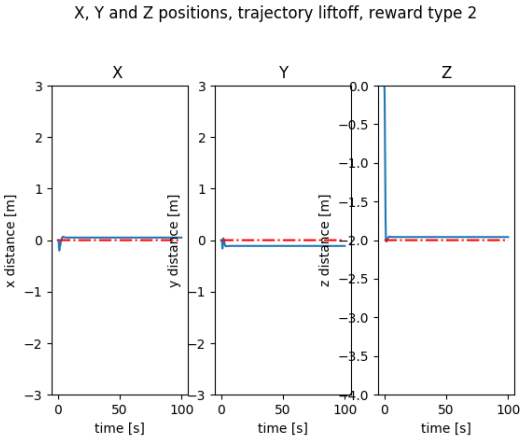
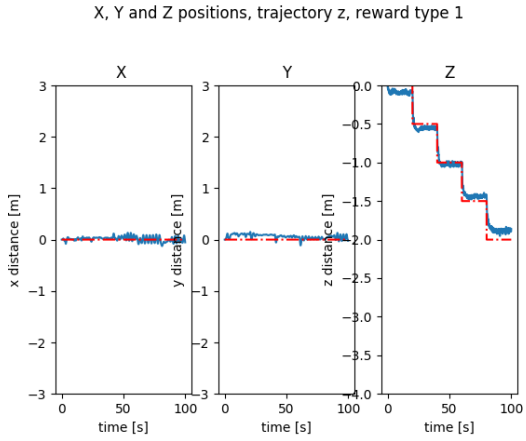


Figure A.1: XYZ positions, liftoff trajectory



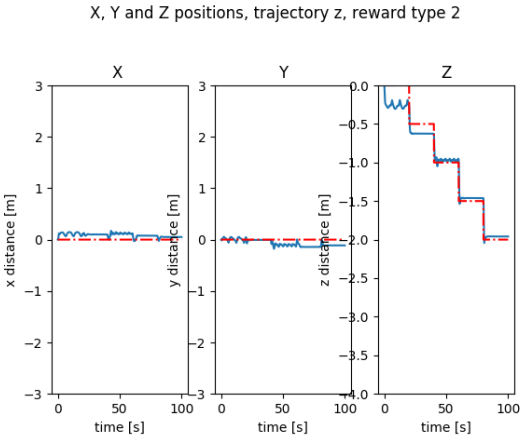


Figure A.2: XYZ positions, z trajectory

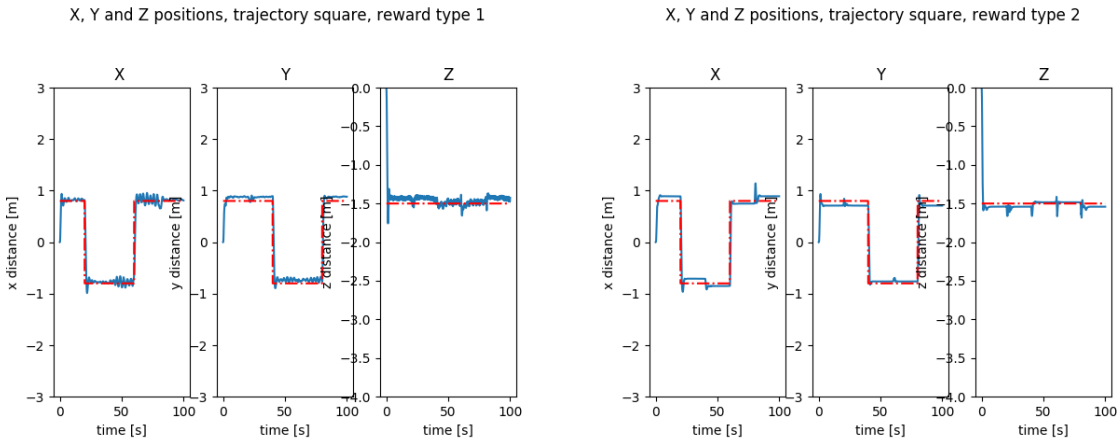


Figure A.3: XYZ positions, square trajectory

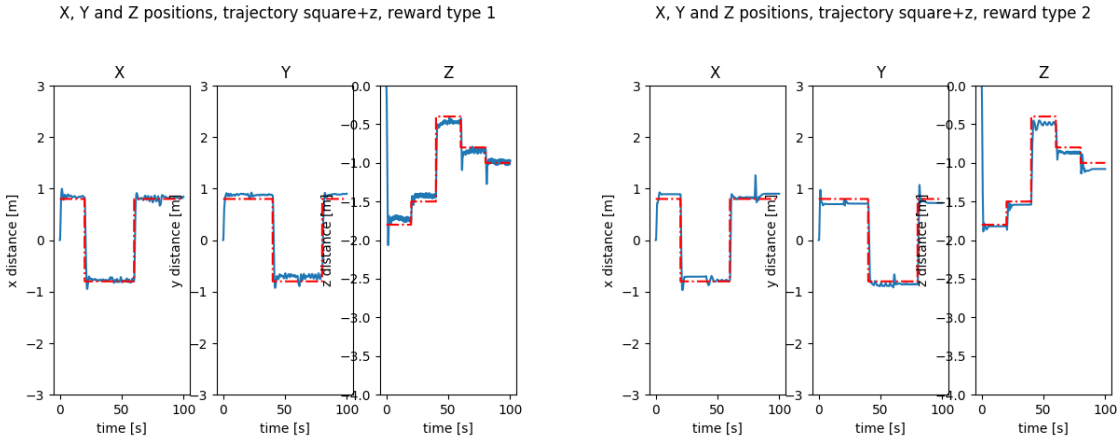


Figure A.4: XYZ positions, square+z trajectory

A.2. Attitude plots

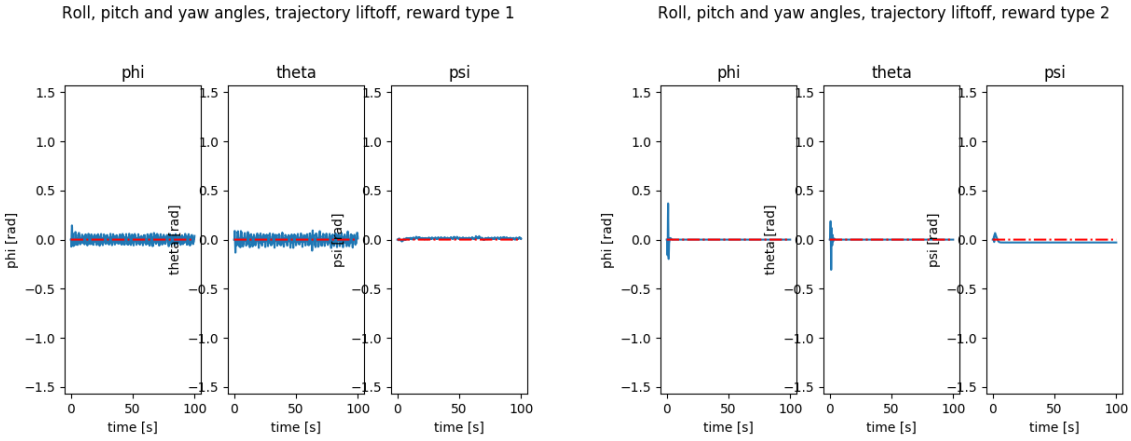
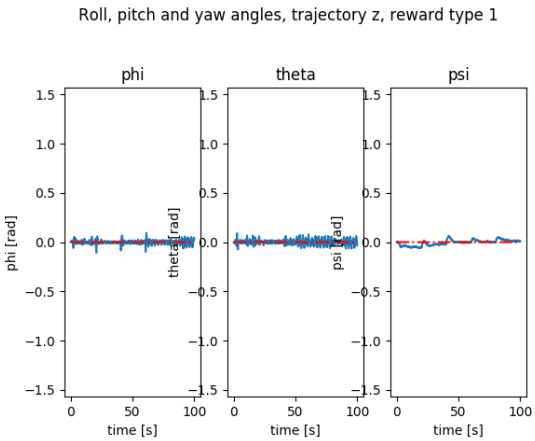


Figure A.5: Attitude, liftoff trajectory



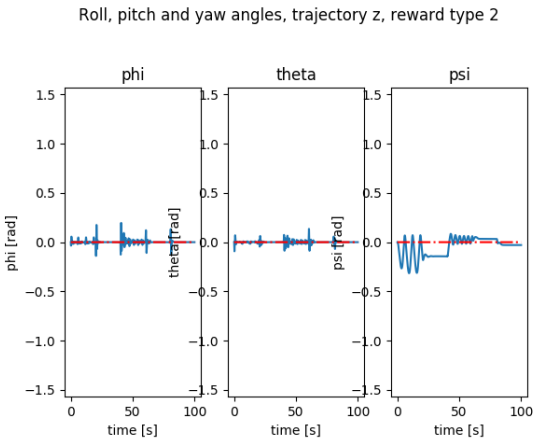
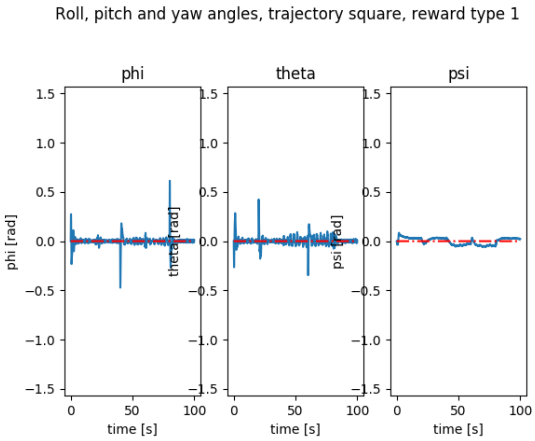


Figure A.6: Attitude, z trajectory



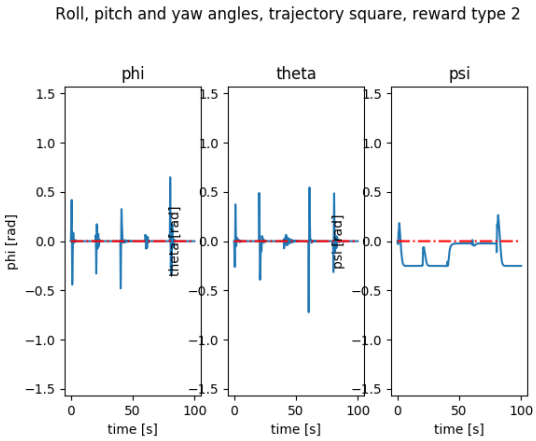
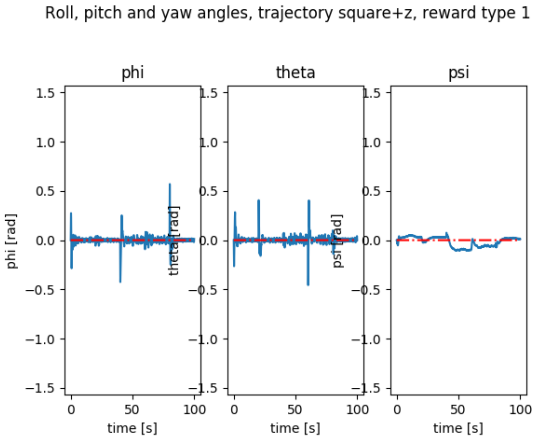


Figure A.7: Attitude, square trajectory



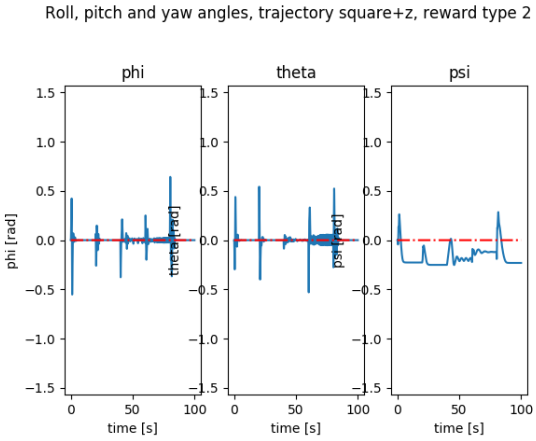


Figure A.8: Attitude, square+z trajectory

A.3. Motor command plots

The figures showing the motor commands of the *type 1* policy are all red. This is because extreme oscillations occur between minimum and maximum motor speeds.

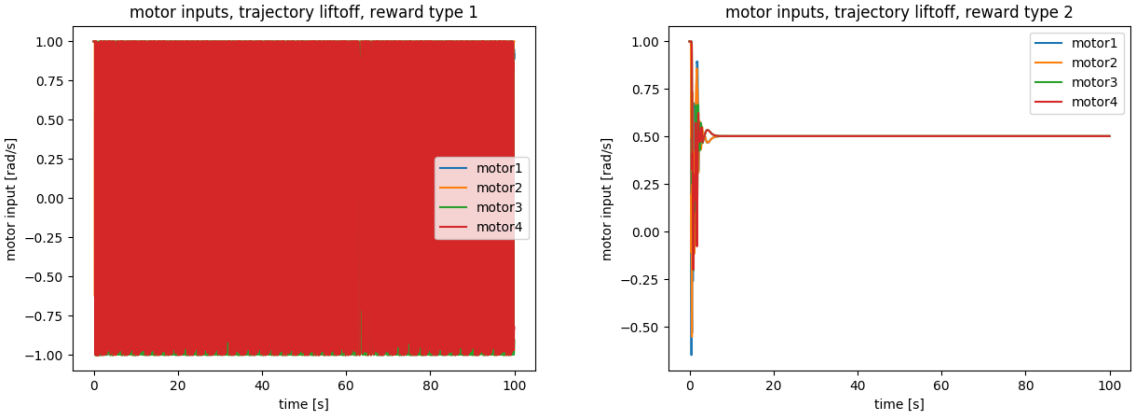


Figure A.9: Motor commands, liftoff trajectory

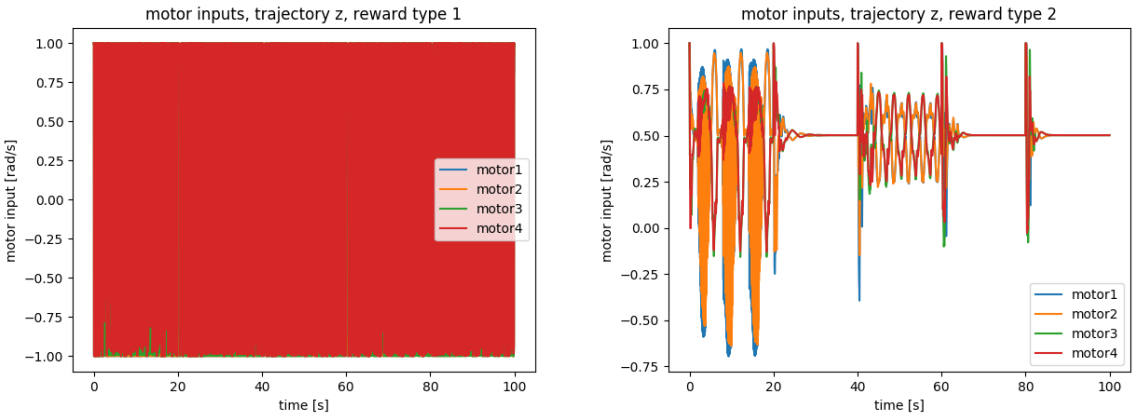


Figure A.10: Motor commands, z trajectory

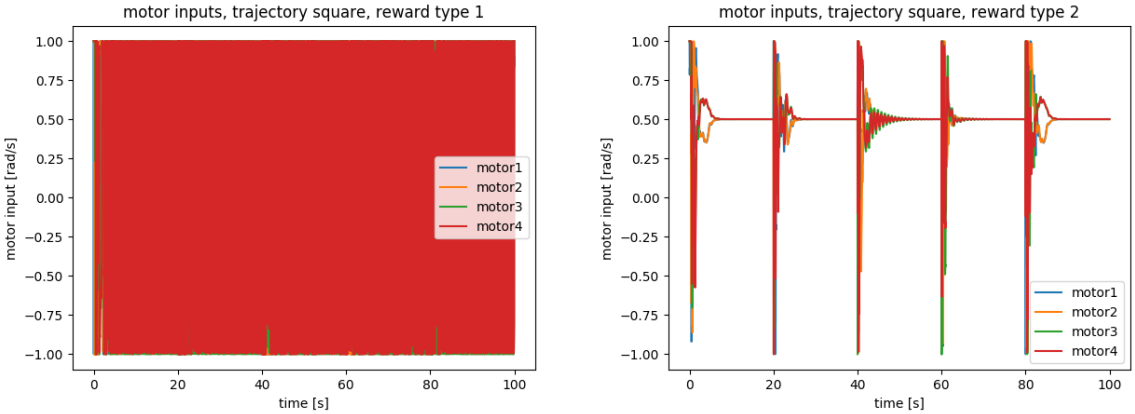


Figure A.11: Motor commands, square trajectory

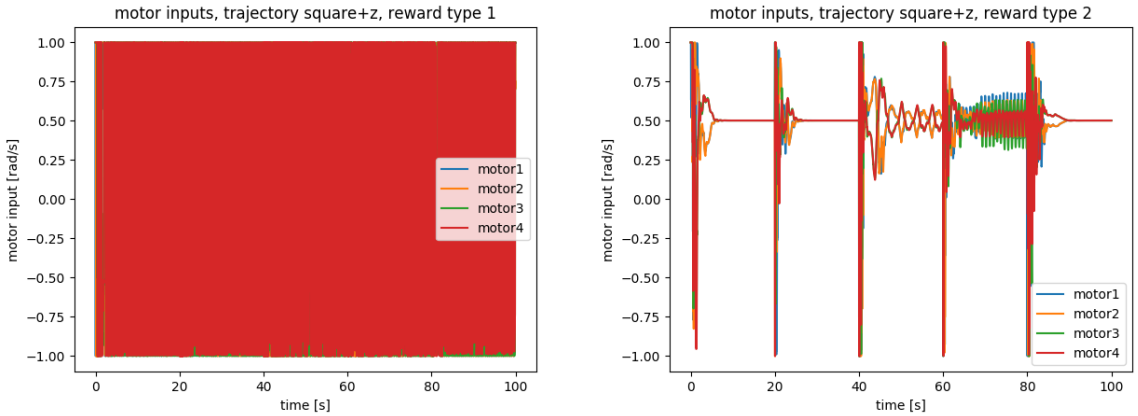


Figure A.12: Motor commands, square+z trajectory

A.4. Position error plots

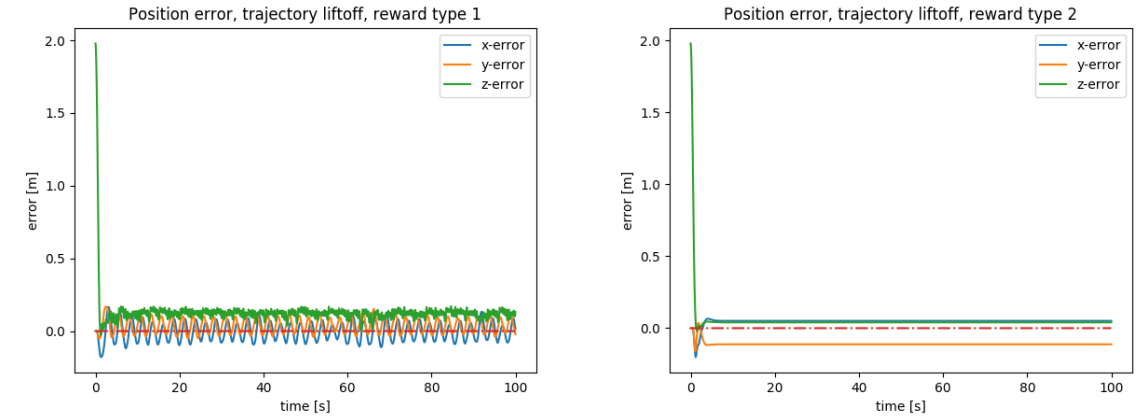


Figure A.13: XYZ position error, liftoff trajectory

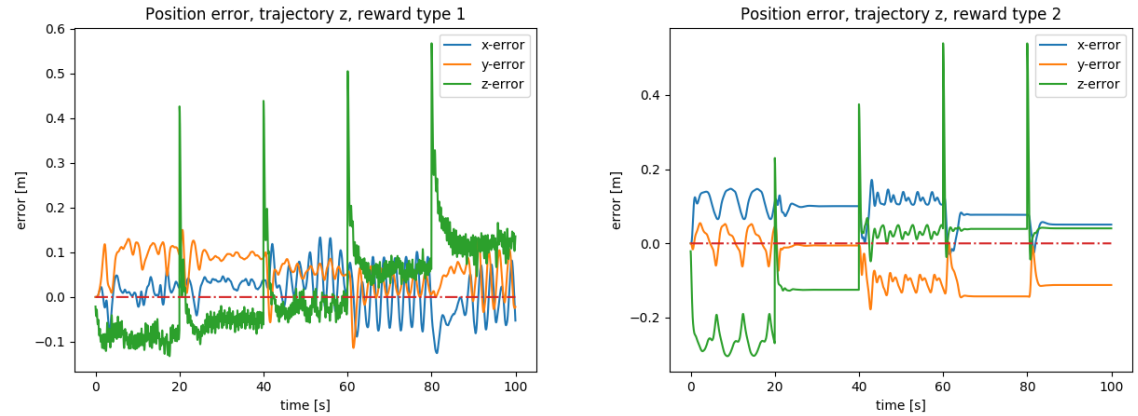


Figure A.14: XYZ position error, z trajectory

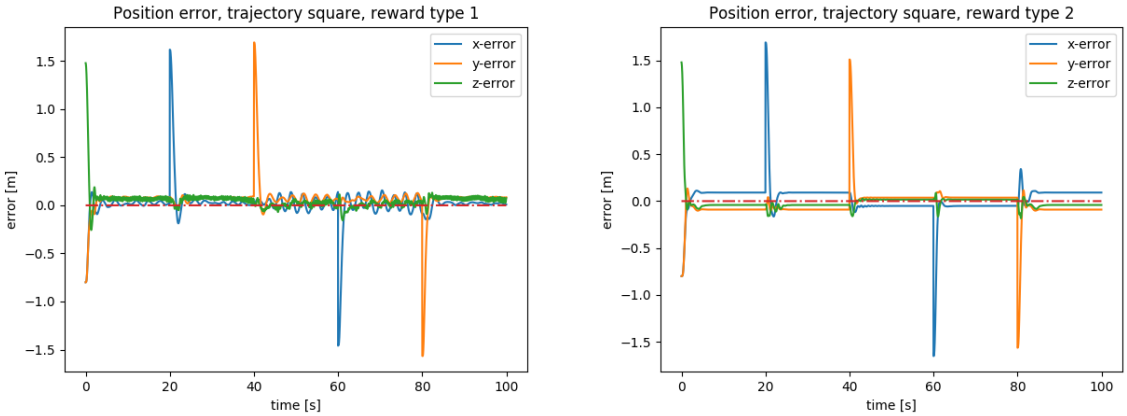


Figure A.15: XYZ position error, square trajectory

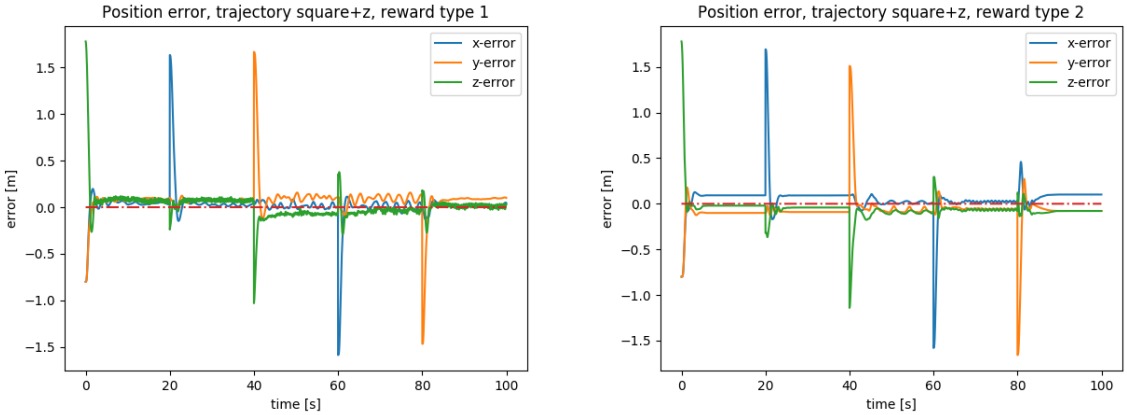


Figure A.16: XYZ position error, square+z trajectory

B

Equations

Rotation matrix

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\psi & -sin\psi \\ 0 & sin\psi & cos\psi \end{bmatrix} \tag{B.1}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix} \tag{B.2}$$

$$R_z(\phi) = \begin{bmatrix} cos\phi & -sin\phi & 0 \\ sin\phi & cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{B.3}$$

Quadcopter dynamics

$$\begin{cases} F_x = m(\dot{u} + qw - rv) \\ F_y = m(\dot{v} - pw + ru) \\ F_z = m(\dot{w} + pv - qu) \\ M_x = \dot{p}I_x + qrI_y + qrI_z \\ M_y = \dot{q}I_y + prI_x - prI_z \\ M_z = \dot{r}I_z - pqI_x + pqI_y \end{cases} \tag{B.4}$$

$$\begin{cases} -mg(sin\theta) + f_{wx} = m(\dot{u} + qw - rv) \\ mg(cos\theta sin\phi) + f_{wy} = m(\dot{v} - pw + ru) \\ mg(cos\theta cos\phi) + f_{wz} - F_t = m(\dot{w} + pv - qu) \\ \tau_x + \tau_{wx} = \dot{p}I_x = qrI_y - prI_z \\ \tau_y + \tau_{wy} = \dot{q}I_y + prI_x - prI_z \\ \tau_z + \tau_{wz} = \dot{r}I_z - pqI_x + pqI_y \end{cases} \tag{B.5}$$

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{u} \\ \dot{v} \\ \dot{w} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} w(\phi\psi + \theta) - v(\psi - \phi\theta) + u \\ v(1 + \phi\psi\theta) - w(\phi - \psi\theta) + u(\psi) \\ w - u(\theta) + v(\phi) \\ rv - qw - g(\theta) + \frac{f_{wx}}{m} \\ pw - ru - g(\phi) + \frac{f_{wy}}{m} \\ qu - pv + g + \frac{f_{wz} - F_t}{m} \\ p + r\theta + q(\phi\theta) \\ q - r\phi \\ r + q\phi \\ \frac{I_y - I_z}{I_x} rq + \frac{\tau_x + \tau_{wx}}{I_x} \\ \frac{I_z - I_x}{I_y} pr + \frac{\tau_y + \tau_{wy}}{I_y} \\ \frac{I_x - I_y}{I_z} pq + \frac{\tau_z + \tau_{wz}}{I_z} \end{bmatrix} \tag{B.6}$$

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{u} \\ \dot{v} \\ \dot{w} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \\ -g(\theta) + \frac{f_{wx}}{m} \\ g(\phi) + \frac{f_{wy}}{m} \\ \frac{f_{wz} - F_t}{m} \\ p \\ q \\ r \\ \frac{\tau_x + wx}{I_x} \\ \frac{\tau_y + wy}{I_y} \\ \frac{\tau_z + wz}{I_z} \end{bmatrix} \tag{B.7}$$

Actuator dynamics

$$\begin{bmatrix} F_{thrust} \\ \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} k_t & k_t & k_t & k_t \\ -lk_t & lk_t & lk_t & -lk_t \\ lk_t & -lk_t & lk_t & -lk_t \\ -k_\tau & -k_\tau & k_\tau & k_\tau \end{bmatrix} \begin{bmatrix} \Omega_1^2 \\ \Omega_2^2 \\ \Omega_3^2 \\ \Omega_4^2 \end{bmatrix} \tag{B.8}$$

C

Parameter estimation plots

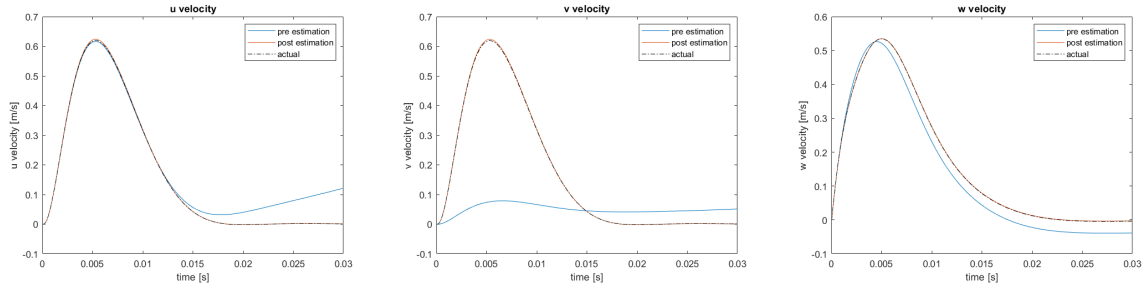


Figure C.1: u, v and w velocities

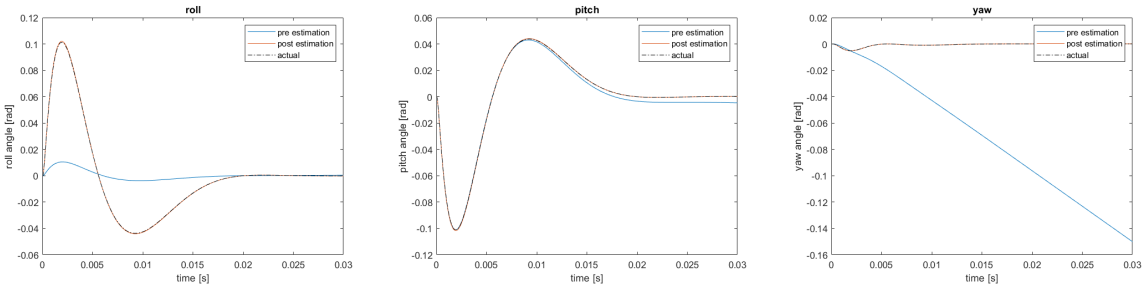


Figure C.2: pitch, roll and yaw angles

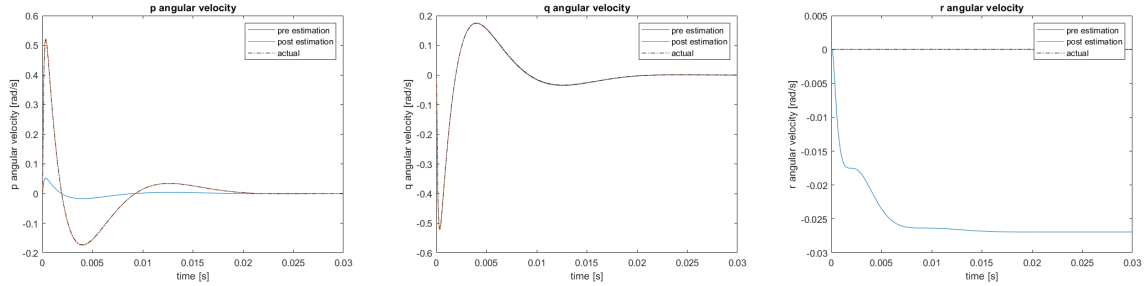


Figure C.3: p, q and r angular velocities

Bibliography

[1] Grey-Box System Identification of a Quadrotor Unmanned Aerial Ve- hicle Qianying Li.

[2] (PDF) Speeding up Deep Learning Computational Aspects of Machine Learning. URL https://www.researchgate.net/publication/308414212_Speeding_up_Deep_Learning_Computational_Aspects_of_Machine_Learning.

[3] Sanjukta Aich, Chahat Ahuja, Tushar Gupta, and P. Arulmozhivarman. Analysis of ground effect on multi-rotors. *2014 International Conference on Electronics, Communication and Computational Engineering, ICECCE 2014*, pages 236–241, 2014. doi: 10.1109/ICECCE.2014.7086619.

[4] Murray Campbell, A. Joseph Hoane, and Feng Hsiung Hsu. Deep Blue. *Artificial Intelligence*, 2002. ISSN 00043702. doi: 10.1016/S0004-3702(01)00129-1.

[5] Timothy M. Cullen. The MQ-9 Reaper remotely piloted aircraft: Humans and machines in action. *Dissertation Abstracts International Section A: Humanities and Social Sciences*, 2014.

[6] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A Survey on Policy Search for Robotics. *Foundations and Trends R in Robotics*, 2(2):1–141, 2013. doi: 10.1561/2300000021. URL https://spiral.imperial.ac.uk:8443/bitstream/10044/1/12051/7/fnt_corrected_2014-8-22.pdf.

[7] Hany R Dwidar. Feedback Optimal Control for Inverted Pendulum Problem by Using the Generating Function Technique. Technical Report 11, 2014. URL www.ijacsa.thesai.org.

[8] Rein Fris. The Landing of a Quadcopter on Inclined Surfaces using Reinforcement Learning. Technical report. URL http://repository.tudelft.nl/.

[9] Scott Fujimoto, Herke Van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. In *35th International Conference on Machine Learning, ICML 2018*, 2018. ISBN 9781510867963.

[10] Marcus Greiff. Modelling and Control of the Crazyflie Quadrotor for Aggressive and Autonomous Flight by Optical Flow Driven State Estimation. page 153, 2017. URL http://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8905295&fileOId=8905299.

[11] V Heidrich-Meisner, M Lauer, C Igel, and M Riedmiller. Reinforcement Learning in a Nutshell. Technical report. URL https://christian-igel.github.io/paper/RLiaN.pdf.

[12] Samuel H. Huddleston and Gerald G. Brown. Machine learning. In *Informs Analytics Body of Knowledge*. 2018. ISBN 9781119505914. doi: 10.1002/9781119505914.ch7.

[13] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. Technical report, 2019. URL http://robotics.sciencemag.org/.

[14] Balakrishnan Jaganatha Pandian, Seshachalam Tharun Kumar, and Mathew Mithra Noel. Information and Control ICIC International c □2017 ISSN. Technical Report 5, 2017.

[15] Łukasz Kaiser, Mohammad Babaeizadeh, Piotr Miło´smiło´s, Bła˙ Zej Osí, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model Based Reinforcement Learning for Atari. Technical report. URL https://goo.gl/itykP8.

[16] Paulin Kantue and Jimoh Olarewaju Pedro. *Grey-box modelling of an Unmanned Quadcopter during Aggressive Maneuvers*. ISBN 9781538644447. URL http://folk.ntnu.no/skoge/prost/proceedings/ICSTCC-2018-Sinaia/ICSTCC2018/papers/0044.pdf.

[17] Derya Kaya and Ali Türker KUTAY. Aerodynamic Modeling and Parameter Estimation of a Quadrotor Helicopter AIAA Atmospheric Flight Mechanics Conference. doi: 10.2514/6.2014-2558. URL http://arc.aiaa.org.

[18] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. 9 2016. URL http://arxiv.org/abs/1609.04836.

[19] S. Lateran, M. F. Sedan, A. S.M. Harithuddin, and S. Azrad. Development of unmanned aerial vehicle (UAV) based high altitude balloon (HAB) platform for active aerosol sampling. In *IOP Conference Series: Materials Science and Engineering*, 2016. doi: 10.1088/1757-899X/152/1/012018.

[20] Dong Bin Lee, Timothy C. Burg, Darren M. Dawson, Dule Shu, Bin Xian, and Enver Tatlicioglu. Robust tracking control of an underactuated quadrotor aerial-robot based on a parametric uncertain model. *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, (October):3187–3192, 2009. ISSN 1062922X. doi: 10.1109/ICSMC.2009.5346158.

[21] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. □□ Google Deepmind 2016 Continuous control with deep reinforcement learning. *4th International Conference on Learning Representations (ICLR 2016)*, 2016. URL http://arxiv.org/abs/1509.02971v6.

[22] Donali W Marquardt. AN ALGORITHM FOR LEAST-SQUARES ESTIMATION OF NONLINEAR PARAMETERS*. Technical Report 2, 1963. URL http://www.siam.org/journals/ojsa.php.

[23] Andreas B. Martinsen and Anastasios M. Lekkas. Straight-Path Following for Underactuated Marine Vessels using Deep Reinforcement Learning. *IFAC-PapersOnLine*, 51(29):329–334, 1 2018. ISSN 24058963. doi: 10.1016/j.ifacol.2018.09.502.

[24] A S Mendes, E Van Kampen, B D W Remes, and Q P Chu. Determining moments of inertia of small UAVs: A comparative analysis of an experimental method versus theoretical approaches. 2012. doi: 10.2514/6.2012-4463. URL http://arc.aiaa.org.

[25] V. Mistier, A. Benallegue, and N. K. M'Sirdi. Exact linearization and noninteracting control of a 4 rotors helicopter via dynamic feedback. *Proceedings - IEEE International Workshop on Robot and Human Interactive Communication*, pages 586–593, 2001. doi: 10.1109/ROMAN.2001.981968.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. 12 2013. URL http://arxiv.org/abs/1312.5602.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015. ISSN 14764687. doi: 10.1038/nature14236.

[28] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving Rubik's Cube with a Robot Hand. 10 2019. URL http://arxiv.org/abs/1910.07113.

[29] Eswarmurthi Gopalakrishnan Prague. QUADCOPTER FLIGHT MECHANICS MODEL AND CONTROL ALGORITHMS. Technical report, 2016. URL https://core.ac.uk/download/pdf/47181924.pdf.

[30] Nan Rosemary Ke, Amanpreet Singh, Ahmed Touati, Anirudh Goyal, Yoshua Bengio, Devi Parikh, and Dhruv Batra. LEARNING DYNAMICS MODEL IN REINFORCEMENT LEARNING BY INCORPORATING THE LONG TERM FU-TURE. Technical report. URL https://arxiv.org/pdf/1903.01599.pdf.

[31] Inkyu Sa and Peter Corke. System identification, estimation and control for a cost effective open-source quadcopter. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 2202–2209, 2012. ISSN 10504729. doi: 10.1109/ICRA.2012.6224896.

[32] Francesco Sabatino. Quadrotor control: modeling, nonlinear control design, and simulation. Technical report, 2015. URL https://www.kth.se/polopoly_fs/1.588039.1550155544!/ThesisKTH-FrancescoSabatino.pdf.

[33] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 1 2016. ISSN 14764687. doi: 10.1038/nature16961.

[34] Gregory G Slabaugh. Computing Euler angles from a rotation matrix. Technical report. URL http://www.gregslabaugh.net/publications/euler.pdf.

[35] Amarnag Subramanya and Jeff Bilmes. Soft-Supervised Learning for Text Classification. Technical report, 2008. URL https://aclweb.org/anthology/D08-1114.

[36] Sihao Sun, Rudi Schilder, and Coen C. de Visser. Identification of Quadrotor Aerodynamic Model from High Speed Flight Data. In *2018 AIAA Atmospheric Flight Mechanics Conference*, Reston, Virginia, 1 2018. American Institute of Aeronautics and Astronautics. ISBN 978-1-62410-525-8. doi: 10.2514/6.2018-0523. URL https://arc.aiaa.org/doi/10.2514/6.2018-0523.

[37] Richard S Sutton and Andrew G Barto. ****Complete Draft****. Technical report, 2018. URL http://incompleteideas.net/book/bookdraft2018jan1.pdf.

[38] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. 3 2017. URL http://arxiv.org/abs/1703.06907.

[39] Mizouri Walid, Najar Slaheddine, Aoun Mohamed, and Bouabdallah Lamjed. *Modelling, Identification and Control of a Quadrotor UAV*. ISBN 9781538653050.

[40] K Wang X. Zhang X. Li and Y Lu. A survey of Modelling and Identification of Quadrotor Robot. Abstract and Applied Anlysis. *Abstract and Applied Analysis*, 2014, 2014.