RAM ROBOTICS AND MECHATRONICS

MODELING, SIMULATION AND CONTROL OF A SET-UP BUILT TO STUDY FLAPPING MOTION

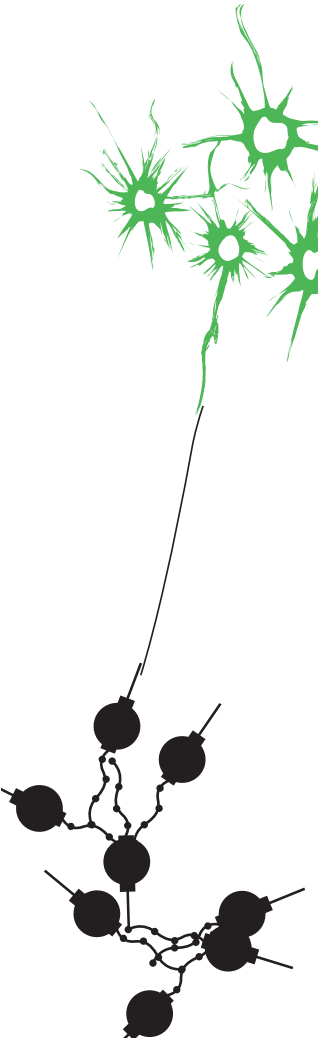
G.H.M. (Gijs) van Rhijn

MSC ASSIGNMENT

Committee:
prof. dr. ir. S. Stramigioli
dr. ir. F. Califano
ir. R.S.M. Sneep
dr. ing. G. Englebienne
dr. E. Mocanu

October, 2022

048RaM2022
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands



UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Abstract

As part of the Portwings project, the Robotics and Mechatronics group at the University of Twente developed a 2 degree of freedom flapping wing setup that is designed to be placed in a windtunnel. This setup is capable of providing real time force and torque information using an advanced sensor. Each axis is independently actuated using electronic motors. The goal of this setup is to deepen the understanding of the unsteady dynamics of flapping flight, which is currently not well understood.

With the creation of such a setup, the question remains how to control each axis to produce an efficient flapping behaviour. This report explores techniques that can be used to optimise such flapping behaviour in a model free way, using experience directly from the windtunnel setup.

Initial attempts at optimisation are made using an actor critic approach from reinforcement learning; here, problems are identified pertaining to reward sparsity, and effective ways encode policy constraints such that it produces a stable and flapping motion.

This inspires a redefinition of the policy in terms of the Fourier decomposition of a flapping trajectory; which automatically encodes the need of a periodic motion, and takes our problem away from the formal reinforcement learning definition and towards a conceptually simpler model free parameter optimisation. Optimisation over this policy is done using the actor critic structure from the original reinforcement learning problem. Experimentally it is shown that this algorithm works for simple reward functions, but that it struggles to optimise over more complicated rewards.

Acknowledgement

First, I would like to express my gratitude to Dr. Federico Califano for giving me the chance to work as part of the Portwings project, and to Prof. Stefano Stramigioli for acting as the chair of my assessment committee.

This project would not have been possible without the regular advice and feedback from my supervisors Dr. Federico Califano and Ir. Riccardo Sneep; their expertise helped me to understand and to grow.

Furthermore, I would like to thank my external committee members Dr. Gwenn Englebienne and Dr. Elena Mocanu for their time and expertise.

Lastly, I want to thank my family and friends for their advice, love and support during this process.

Contents

1 Introduction

After developing a working flapping autonomous drone, the Robotics and Mechatronics (RaM) research group at the University of Twente (UT) now aims to deepen our understanding of flapping flight in the form of the Portwings project [6]. Currently the unsteady aerodynamics of flapping flight are not well understood. The project aims to create a 'much deeper structured understanding of flapping flight' through the use of port-Hamiltonian modelling techniques to model these dynamics, and to do finetuning and verification of these models by way of experiment.

Part of the Portwing's projects efforts to improve our understanding of flapping flight is the development of a wind tunnel setup capable of 2 degrees of freedom (DoF) flapping behaviour. The setup provides real-time feedback through a 6 dimensional force sensor at the base of the wing.

Each axis of the setup can be actuated independently through the use of two separate electric motors. A controller is needed to ensure useful, stable flapping behaviour. This report aims to explore the question how to optimise flapping flight on this setup, either through directly learning a useful feedback law on the motor torques, or by means of trajectory optimisation in combination with a state feedback controller.

Currently, there is no known best way to optimise a problem of this type. Therefore, the report will take on an exploratory character. It will explore and discuss the merits and limitations of various techniques, both in theory and by way of simulation and experiments.

It is important to note that full understanding of the aerodynamics of flapping flight remains a unsolved problem and an active field of research; therefore this report will focus on model free techniques that allow the use of the actual, physical setup for the optimisation.

1.1 Research questions

The main research question designates the focus of this report to be on the optimisation of flapping behaviour; where an optimal flap is considered to: generate desired dynamic effect, and to do so in an energy efficient manner. This energy efficiency is considered by the researchers to be a fundamental part of what what it means for a flap to be optimal.

Furthermore, the research question includes the need to take advantage of the physical flapping setup. The motivation behind this being the lack of current understanding of the aerodynamics of flapping flight, in combination with the unique opportunity to learn from real – not modelled – dynamics.

Main question:

- **How can we optimise a flapping motion in terms of the energy efficient generation of desired effects, while taking advantage of a physical flapping setup?**

The 2 sub-questions refer to 2 possible techniques that could be used to solve the optimisation problem of the main question. The report is divided into 2 parts, one for each question, with a unifying discussion and conclusion at the end. The motivation to ask the second question comes from the research done on the first; which makes it not only advisable to read the two parts of the report in order, but it also shows the exploratory nature of the project.

Sub questions:

- **Can we use continuous reinforcement learning techniques to directly learn a state feedback policy that solves the optimisation problem?**

- **Can we repurpose the actor critic architecture from reinforcement learning to optimise over Fourier coefficients representing a flapping trajectory?**

1.2 Defining optimality

As indicated by the main research question, the optimality of the flapping behaviour is defined to be in terms of the energy efficient generation of desired effects. This is mathematically expressed in terms of a ratio between the quantity of desired effects generated during a flapping period, and the energy used during that same period:

$$\frac{\int_{<T>} h(t)dt}{\int_{<T>} p(t)dt} = \frac{H}{E} \tag{1.1}$$

In this equation the desired effects at a point in time are indicated by $h(t)$, while $p(t)$ indicates the power usage. The expression $< T >$ indicates that the integral is defined over a single period of the flapping motion, which means that it can only be calculated after the entire period has finished

An alternative way to quantify the energy efficient generation of effects during a flapping period would be to use a subtraction where the energy use is subtracted from the desired effects:

$$\alpha H - \beta E \tag{1.2}$$

But the meaning of it is not clear, and the optimum can be easily changed by a linear change of unit choice. For this reason, the Equation 1.2 is not used, instead the reward solely focuses on optimising directly over Equation 1.1

1.3 Structure of the report

The report is divided into 2 parts:

The first part (Chapter 4), which refers to the first sub-question and explores the possibility of using reinforcement learning to solve the optimisation problem. Here problems are identified that pertain to the sparsity of the reward, and the difficulties of learning flapping behaviour purely through the experience of a state feedback controller that controls the motor torques of the setup.

The second part (Chapter 5), which tries to solve the problems identified in the first part through a redefinition of the policy using the Fourier decomposition of the trajectory, and the resulting optimisation algorithm. This is a exploration of the second sub-question.

Before the two sub-questions are explored there is a short chapter on the feasibility of using optimal control to help solve the optimisation problem (Chapter 2). The information from this chapter is not used in the rest of the report, but it is left in anyway as it fits the exploratory character of the report.

Additionally there is a chapter describing the setup, a placeholder setup that is used during the experiments because the final setup was not available yet, and a digital model of the placeholder setup (Chapter 3).

Finally, Chapter 6 discusses the results of the whole report, Chapter 7 provides an overall conclusion, and Chapter 8 gives an overview of different insights and alternative approaches in literature that could help solve some of the problems encountered in the report. It also provides the reader with a final conclusion.

2 Optimal control

Before taking a look at the model free techniques in chapters 4 and 5, this chapter will quickly discuss the possibility of using optimal control techniques to gain insight into the optimisation problem. Optimal control techniques are model based. This means that they will require the use of a dynamic model for optimisation, and cannot take direct advantage of a real setup. The information in this chapter is not needed to understand the rest of the report as it was decided not to continue with these techniques. Instead this chapter was kept in as it fits the exploratory nature of the report, and it could be informative to subsequent research.

2.1 Techniques

Optimal control as a discipline differs from traditional control by attempting to find a (globally) optimal solution as defined by some cost/reward function. The cost functions used in optimal control is often expected to be of the form:

$$J(u) = \int_0^T L(x(t), u(t))\,dt + K(x(T)) \tag{2.1}$$

Here, the L represents the running cost, which is dependent on the states $x(t)$, and inputs $u(t)$ from the initial time $t = 0$ till the final time $t = T$. The final state of the setup is penalised using the terminal cost $K(x(T))$, which is dependent on the final state $x(T)$. Setup states throughout the trajectory are constrained by the system dynamics, generally expressed as a set of ordinary differential equations (Equation 2.2), and input limits (Equation 2.3): [24, 9]

$$\dot{x} = f(x(t), u(t)), \quad x(0) = x_0 \tag{2.2}$$

$$u : [0, T] \rightarrow \mathbb{U} \tag{2.3}$$

Classic optimal control theory makes it possible to find an analytical, guaranteed globally optimal solution to a problem [24]; however, results in classical optimal control theory can be difficult to apply in practice, and the classes of problems that have reliable solution methods are limited.

Analytical techniques in optimal control include the Euler Lagrange equation and Pontryagins minimum principle, which try to find an optimum via the use of necessary conditions; and dynamic programming: effectively running through a problem backwards and keeping track of the most optimal trajectory from each state till the end point [24].

Instead, numerical methods where devised that aim to take advantage of the results from optimal control theory, but make it easier to find something close to an optimal solution [18, 12, 9]. Numerical methods can try to directly solve the problems mentioned above (indirect methods and numerical dynamic programming), but what is generally a more successful approach is to discretize the dynamics and reformulate the problem into a non-linear programming problem [12, 9]; which can be solved using specialised numerical solvers. This family of approaches is often referred to as "direct methods", while Euler Lagrange and Pontryagin based methods are referred to as "indirect methods".

While the non-linear programming and indirect solution methods result in global, state feedback solutions to the problem, direct solution methods only provide a single optimal trajectory, specific to it's initial conditions.

2.2 Applying optimal control to the setup

When applying these techniques to the optimisation problems in this report, a few limitations can be identified:

Firstly, optimal control theory is designed to optimise over a known cost function and a known set of setup dynamics; therefore, the setup dynamics have to be expressed analytically during the problem formulation. Flapping wing dynamics are not easily transcribed into simple mathematical relations, which is one of the reasons the physical setup exists. Optimal control techniques do not allow us to take direct advantage of the physical setup, and it's real time sensor data.

Secondly, the most commonly used optimal control techniques – the direct methods – do not result in a globally defined solution to the problem, which makes it not straightforward to translate them to a full control law that is implementable on the real setup.

These difficulties with applying optimal control techniques to the optimisation problem of this report have lead to the decision to not use them in the experiments. Instead, the focus is on the use of model free optimisation techniques, see Chapter 4 and 5.

3 (Physiscal) Experimental setup

3.1 Full setup

This report was started to take advantage of a flapping wing setup currently in development at the Robotics and Mechatronics (RaM) department of the University of Twente (shown in Figure 3.1). This setup is designed to be used as a research platform for flapping flight. Therefore it is equipped with advanced sensors and designed to be placed into a wind tunnel. The setup has 2 independently actuated degrees of freedom (DoF) denoted by q_1 and q_2. (Figure 3.2)

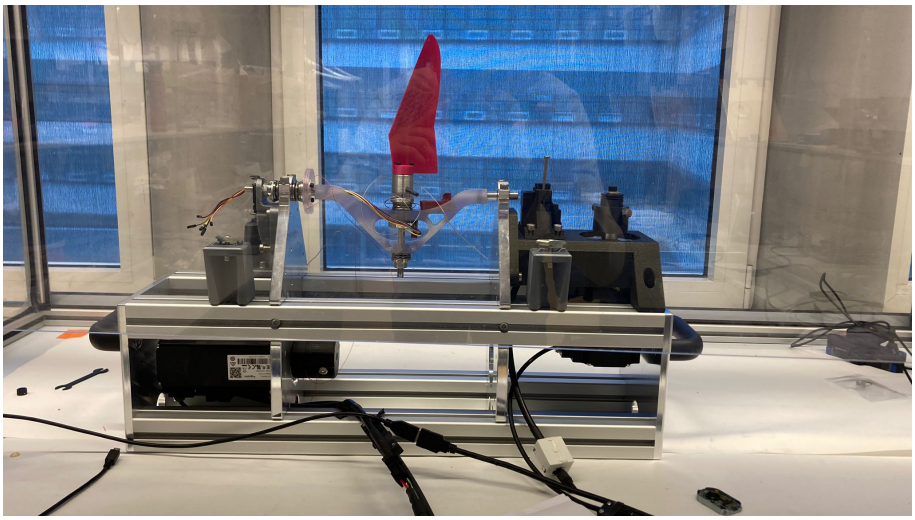


Figure 3.1: Image of the full flapping wing setup with a small red test wing attached.

Each axis is independently connected to electric motors via a cable drive. These cable drives allow the motors to be placed away from the wing during operation; which is specifically useful for placement of the setup in the windtunnel. They do however introduce elasticity to the actuation mechanism. The output torques of the motors are the input of the system.

Each motor is directly connected to a rotatory encoder providing angle measurements of the motor axle. Angle measurement of the wing axle on the other side of the cable drive are not available. Velocity measurements are not available at all, but could be estimated based on the position measurements if required.

Measurements from the wing are available through a 6 DoF sensor placed in between the wing and the rotating axis; providing us with the full wrench from the wing on to the frame. These measurements together with the position measurements from the rotary encoders, constitute the full measurable available state information. Other state variables can be estimated from the available measurements, ex. motor velocity, or are simply unavailable, ex. the infinitely dimensional aerodynamic states.

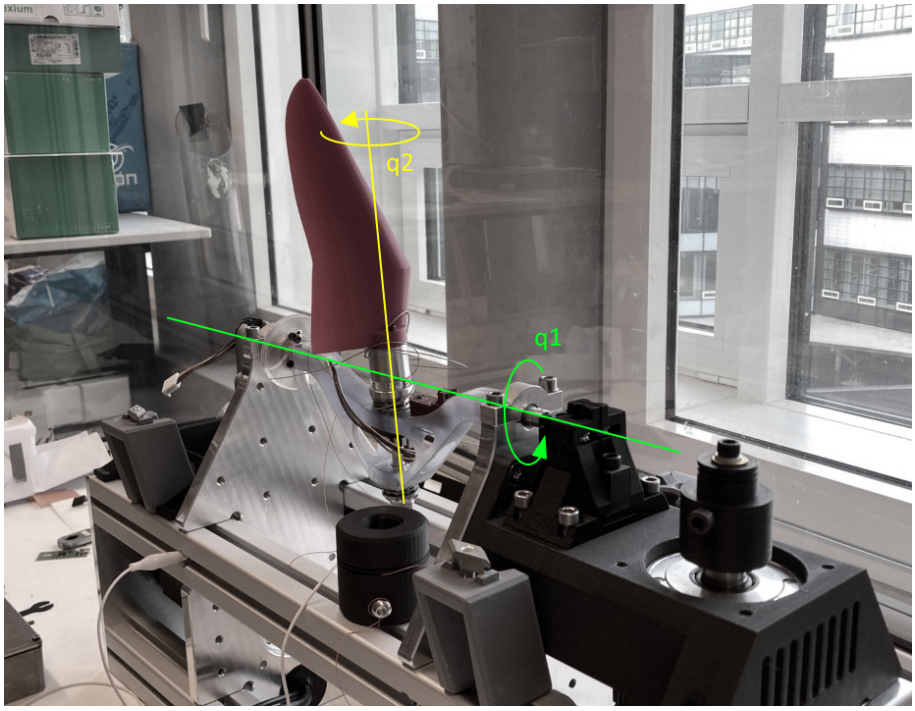


Figure 3.2: The q_1 (green) and q_2 (yellow) rotational axes on the setup.

3.2 Placeholder setup

As of the writing of this report, the full setup is not yet fully functional. Preliminary tests of an optimisation technique can be done using a digital physics simulation. More comprehensive validation of the optimisation technique should be done on a physical setup. For that reason a simpler temporary placeholder setup was constructed (Figure 3.3). This setup is not designed to be used in a windtunnel.

This placeholder setup was constructed to have the same 2 DoF as the original. A Nema 17 servo motor [1] controls axis q_1, and a DFRobot DSS-M15S servo motor [3] controls axis q_2. Both motors are controller by a Mechaduino micro controlled [1].

The wing is a flat sheet of plastic that is 220 cm wide by 200 cm high.

Figure 3.3: Image of the placeholder flapping wing setup surrounded by a protection cardboard cage.

3.2.1 Controlling axis (q_1)

For the experiments in Chapter 5, both axes of the setup need to be equipped with a controller that is able to follow the trajectories. Using the opensource code library provided with the Mechaduino, it is possible to implement a PIDF (F stands for low pass filter) controller on the Nema stepper motor attached to q_1. This library uses the current motor position to estimate the stepper motor input needed to create the torque that is demanded by the controller.

For the purposes of this setup, operational limits are set to $[-180°, +180°]$. The non-linear effects of gravity were cancelled out using position information, linearizing the system. The angle is controlled using of a PIDF controller. The gravity compensation functionality is not native to the Mechaduino and had to be added manually.

Due to the required conversion between the desired controller torque and the stepper motor input, the unit of the control value is not clear, nor is it known if it relates linearly to the output torque of the motor; therefore both the PIDF controller and the gravity compensation where implemented by inspection.

3.2.2 Controlling axis (q_2)

For the experiments in Chapter 5, both axes of the setup need to be equipped with a controller that is able to follow the trajectories. Controlling q_2 using the servo motor is straight forward. The servo motor has a $[-135°, +135°]$ range, allowing it to easily move between a fully open and fully closed wing position. The servo is installed in such a way that the $0°$ angle is placed exactly in between the open an closed wing position. Setting the servo angle is done using the standard Arduino Servo library [5].

3.2.3 Available sensor data

The placeholder setup does not possess a force sensor; instead, the state information is limited to the motor positions (both q_1 and q_2), and the output signal of the control algorithm for q_1. Again, this output has no known unit, nor is it known how linear its relation is to the actual output torque.

To construct the reward function the motor torque of q_1 has to be used, as no other data is available.

3.3 Digital model placeholder setup

To help with testing and evaluation, a digital model of the placeholder setup was created.

The mathematical geometry of the model is discussed in Appendix B. In essence, the setup is a 2 DoF manipulator with an actuator on each joint. Unlike the full setup, the motors are attached directly to the joints; therefore, no series elastic actuation was modelled.

Each motor is modelled as a single torque source controlled by a feedback controller. The controller on axis q_1 uses a combination of gravity compensation and PID control just like the real setup. Reward functions are calculated using the output of this controller, again to mimic the real setup.

The motor controller on q_2 is modelled as a PID controller. No gravity compensation is needed on this axis. The data from this axis is not used for for reward function calculations as no data from this axis is available from the real placeholder setup.

The inertias of the axis are modelled as homogeneous masses of approximately the same shape and size of their real life counterparts. The total weight of each intertia is the same as the real component.

Wing dynamics are not easily modelled and it's not completely obvious what (aero)dynamic effects are present on the flat rectangular wing of the placeholder setup. The placeholder setup is not designed to be used in a windtunnel, which might limit the complicated aerodynamic effects present at the wing. The same can be said for the simple wing shape (flat plate) and the lack of flexibility in the wing.

To avoid unsubstantiated guesses about wing dynamics, they where only modelled as a damping force. The equation used for this is an approximation of the force on a flat plate moving though the air at a given velocity. Since the velocity of the wing is not the same at every point, the equation was integrated to work with a given angular velocity. Lastly, the width of the wing was multiplied by the absolute value of the cosine of q_2 to model the effects of the wing angle:

$$\begin{aligned} |F_{drag}| &= q_1^2 \cdot \frac{\rho \cdot C_D \cdot q_2 \cdot W_{wing}}{2} \cdot |cos(q_2)| \cdot \int_0^l r^2 \cdot dr \\ &= q_1^2 \cdot \frac{\rho \cdot C_D \cdot q_2 \cdot W_{wing}}{6} \cdot |cos(q_2)| \cdot l^3 \end{aligned} \tag{3.1}$$

Here ρ represents the air density, C_D the drag constant, W_{wing} the width of the wing, and l the length.

Simulation is done using the Scipy integrate ODE function [4] in a python environment. This function allows for the numerical integration of ordinary differential equations (ODEs) in state space form.

4 Part 1: Reinforcement learning

4.1 Introduction

In this part of the report the feasibility of using reinforcement learning for solving the proposed optimisation problem will be discussed. First, a summary will be given of the basic reinforcement learning problem definition based on the book by Sutton and Barto [29]. Then, current literature on using reinforcement learning for unmanned areal vehicle (UAV) control will be discussed with regards to the optimisation problem of this report. Next, the challenges of designing a reward function will be discussed, and it is proposed that a direct implementation of the learning algorithm on the problem is unlikely to result in success. This is corroborated by means of an experiment using the deep deterministic policy gradient algorithm (DDPG). The discussion will analyse the results of the experiment, and suggest an alternative task definition which will lead to the optimisation algorithm discussed in part 2 of the report.

4.2 Background

4.2.1 The reinforcement learning problem

Reinforcement learning aims to learn how to interact with an environment in a model free way [29]. Figure 4.1 shows the basic model for interaction between the agent (the element that makes decisions and tries to learn from them) and the environment (the element that the agent interacts with).

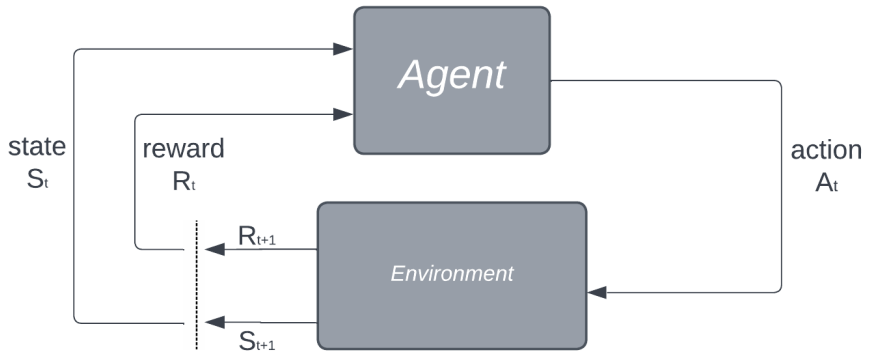


Figure 4.1: Basic schematic of the interaction between the reinforcement learning agent and its environment.

The agent takes the position of a state feedback controller that controls the environment in discrete steps. The probability of taking a certain action given a state is called the policy $\pi(A_t|S_t)$. After each interaction the environment returns a reward and a new state. This leads to a chronological sequence of states, actions, and rewards: $S_0, A_0, R_1, S_1, A_1, R_2, ...$ (where the subscripts indicate the discrete time-step).

Optimisation of the policy is done by maximising the expected future reward: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$, where $0 \leq \gamma \leq 1$ is called the discount rate and represents the balance between short term and long term gains. Reinforcement learning tasks can be defined to continue indefinitely (the sequence of states, actions and rewards never ends), at which point $\gamma < 1$ to avoid G_t becoming infinitely large. The tasks can also be defined to be finite (episodic), which is where the sequence of actions, states, and rewards is stopped at some point, and the decision

process is restarted from some initial position (think for example of a game of chess that ends after check mate). These finite length decision processes are called episodes.

4.2.2 The Markov property

The reinforcement learning update rules assume it is useful to keep track of a value function per state/action pair. Underlying this assumption is the idea that the states provided to the agent are Markov. A state being Markov is defined as the probability of a future state and reward being only dependent on the current state and action, not on any past states/actions [29, Chapter 3]. This ensures the environment provides the agent with the full information to form an optimal policy. In reality, this assumption is often not satisfied and exact proofs of convergence are replaced with empirical conclusions and intuition. This is also the case for the problem of this report due to the infinite dimensional nature of the flapping wing dynamics involved.

4.2.3 State of the art

Using a reinforcement learning agent as a controller for the physical flapping wing setup requires the algorithm to be able to deal with a continuous state and action space. Currently there are 4 commonly used reinforcement learning algorithms that are used for continuous control: TRPO[25], PPO[26], DDPG[22], and SAC[15]: TRPO and PPO can only learn from experience gained under the current policy (on-policy), which makes them potentially less sample efficient. They are also limited to non-deterministic policies. Nevertheless, research shows them to be successful at attitude control for UAVs, specifically with training done in a time efficient simulation environment [20, 10].

On the other hand, DDPG and SAC are off-policy algorithms which allows them to learn from past experience stored in a replay buffer, making them more sample efficient and arguably more suitable for training on a physical setup. This idea is also entertained in the conclusion of [10]. DDPG and SAC also allow for the use of a deterministic policy, which is desirable with regards to the predictability of the control law, and they are used successfully in the control of UAVs [16, 30].

DDPG and SAC uses an actor critic architecture for learning a successful policy. As the name suggests, the actor critic architecture divides the learning algorithm into two parts: the actor, which represents the map from the current state to an action (for DDPG this is a neural network); and the critic (again a neural net) which estimates the future expected reward given the current state and the chosen action.

The critic is trained using supervised learning updates. The target values for these updates have to represent an estimate of the future expected reward given the current state and action. These can be estimated using the Bellmann equation [22, Chapter 2] [15, Chapter 4.2]; which can estimate the future expected reward for a state action pair using past experience and the current critic. Note how the future expected reward changes every-time the policy changes. This makes the target values for the critic network non-stationary, making the training process of the critic network less sample efficient and possibly less stable.

The actor is updated by sampling past states, and calculating the gradient of the Q network output over the network parameters of the actor (π) in those states. This results in a estimate of the gradient of the expected future reward over actor network parameters. The actor network can then be updated in the direction of the greater expected future reward. This gradient is called the policy gradient.

The state of the art in the control of UAVs mostly focuses on the control on quadcopters or fixed wing aircraft. Here control tends to focus on achieving a given attitude, or following a given trajectory. Reward function design tends to use the some combination of the error

between the desired attitude/trajectory and the current setup state of the drone, with extra punishments to encourage smooth controller outputs and energy efficiency. This gives rise to naturally dense rewards, that allow small improvements in performance to be directly expressed: desirable qualities for reinforcement learning problems [23] [29, Chapter 17.4]. On the other hand, the addition of punishments for undesirable effects makes the meaning of a optimal policy harder to interpret.

No research on the model free optimisation of flapping behaviour was found by the author. Differences can be identified between optimising quadcopter or fixed wing flight, and flapping behaviour: firstly the physical setup does not directly represent a full bird in flight, so optimisation cannot be done in terms of trajectory or attitude control. Instead optimisation has to focus on the generation of specific aerodynamic effects that can be generated in a wind tunnel. These effects, if well chosen, could then be used to obtain actual flapping flight. Secondly, flapping flight is inherently periodic and not constant in its generation of effects, and requires the prioritisation of long term rewards over short term rewards. When the reward is expressed in terms of the sparsely defined ratio discussed in Section 1.2, both the need for periodicity and the need for prioritisation of long term effects is not directly communicated to the reinforcement learning agent.

4.2.4 Reward function design

In [23] and [29, Chapter 3.2 and 17.4] reward function design for reinforcement learning agents is discussed. [29, Chapter 3.2] describes the primary purpose of the reward function as describing what the engineer wants the reinforcement learning agent to achieve. In this respect the ratio function discussed in 1.2 looks to be well defined. However, the existence of this reward function is very sparse; it only exists after finishing a full flapping motion. Reward sparsity is identified as a problem for learning complex tasks [23] and [29, Chapter 17.4]. A sparsely returned reward can cause problems with convergence or unexpected behaviour of the actor. Furthermore, for the ratio reward discussed in Section 1.2 the return of a reward is not guaranteed and dependent on the behaviour of the algorithm – if the policy is unstable for example, no reward will ever be returned. This can cause what [29, Chapter 17.4] refers to as the "plateau problem", where the reinforcement learning agent has periods of training time where its behaviour does not lead to any reward.

One solution to the plateau problem – and a way to reduce reward sparsity in general – is to add extra rewards that are supposed to guide the learning algorithm to the optimal policy, without actually affecting the definition of the optimal policy, but all the while providing a useful heuristic to the learning algorithm to help with convergence. In our case, this is done by the way of returning negative rewards (punishments) whenever the policy does not produce a full flapping motion, as described in Section 4.3.2.

4.3 Method

This experiment aims to test the feasibility of training a continuous reinforcement learning agent as a torque feedback law, using the ratio reward function as described in 1.2. The experiment uses the standard DDPG algorithm for learning. The torque output of the motors are defined to be the output of the DDPG agent. In turn, the current position and velocity of the setup are the input of the agent. This gives the DDPG agent the role of a state feedback controller, see Figure 4.2.

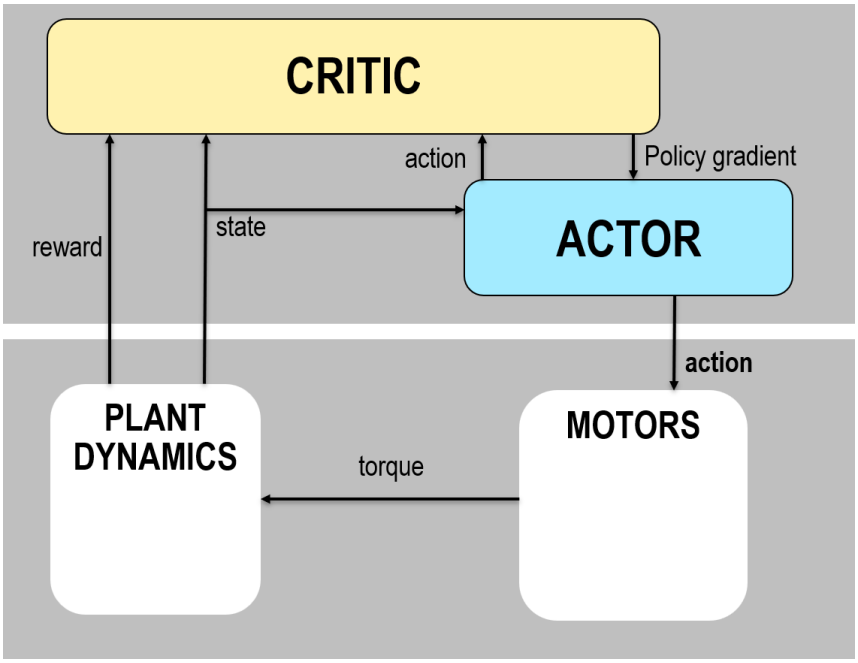


Figure 4.2: The learning scheme of the DDPG algorithm in this experiment. The actor acts as a state feedback controller and the critic provides the policy gradient for learning.

In this experiment, training will be done on a digital dynamic model. The wing- and aerodynamics are not modelled (details in Section 3.3). The reason for this is twofold: Designing an accurate, and reasonably fast running, aerodynamic model of a flapping wing is not feasible, and testing the effects of these dynamics on the Reinforcement learning algorithm is not the point of this experiment. Instead the point is to test the reward function.

The hyperparameters for the DDPG algorithm are mimicked from the recommendations in the original paper [22] (see Table 4.1). The task is defined episodic, and each episode is ended after a random amount of time (varies linearly between 1 and 10 seconds) to prevent the policy from adapting to a specific end time. The experiment is ran for 100,000 timesteps of 0.02 seconds each. Based on the results in the original DDPG paper, we would expect this to be enough timesteps for improvement to show.

Parameter	**Value**	**Parameter**	**Value**
Mass pendulum (kg)	1.0	Learning rate critic	1e-3
Length pendulum (m)	1.0	Learning rate policy	1e-4
Max torque (Nm)	20	Discount rate	0.99
Initial q_1 (rad)	0	Batch size	64
Step size (s)	0.02	target network update rate	0.001
episode lenght (s)	1-10	theta noise	0.15
total number of steps	100,000	sigma noise	0.2
amplitude treshhold (rad)	0.1	scale noise	3
		Layer sizes critic	400, 300
		Layer sizes actor	400, 300

Table 4.1: Model- and hyperparameters of the experiment. The hyperparameters are based upon the recommendations in the original DDPG paper [22].

4.3.1 Digital environment model

The point of this experiment is to test the feasibility of training the reinforcement learning agent using the ratio reward function; therefore the aerodynamic effects of the wing are left out of the digital model. This removes the effects of axis q_2. Therefore this axis is removed from the model. This results in a 1 DoF pendulum model, see Figure 4.3. The mass of the system is represented as a point mass at the end of the pendulum arm.

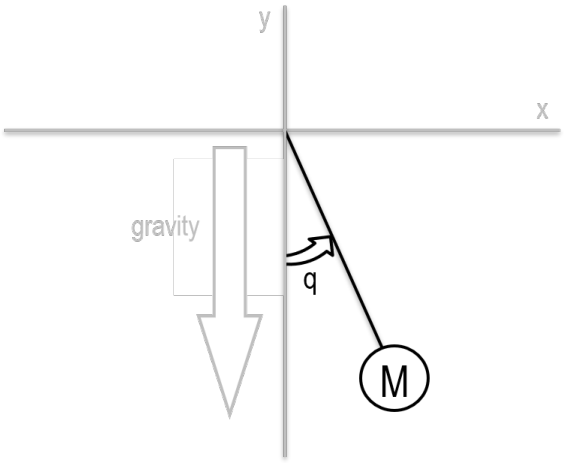


Figure 4.3: Geometry of the pendulum model. Setup mass is modelled as a point mass (M). The stable equilibrium point is at $q_1 = 0$.

Although this model is greatly simplified from the real setup and misses some fundamental dynamics, it contains all the functionality necessary to test the reward function.

While the dynamic effects of the wing are not included in the dynamics model, It is still necessary to define a heuristic for the numerator of the reward function. This is done by integrating the cube of the angular velocity of q_1, multiplied by its original sign. This does not represent any real thrust or lift values but – intuitively – it encourages a wing flap with a fast stroke in the positive direction and a slow stroke in the negative direction, creating a clear optimal policy to work towards and making any final policy easier to interpret.

The energy use is computed by integrating over the input power; which is defined as the angular velocity times the input torque. Since there is no way to recuperate energy on the current setup, the power is defined to always be equal or greater than zero.

Practically, the physics are simulated in Python using the Scipy ODE integrator [4]. This library allows the integration of ODEs in state space form. The policy is called at 50hz. For the full list of DDPG hyper-parameters and model constants, see Table 4.1

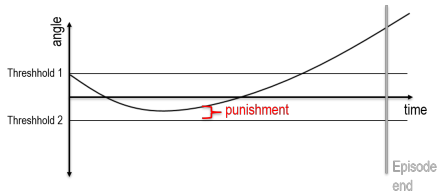
4.3.2 Reward function design

Whenever the policy completes a swinging motion with an amplitude of at least 0.1 radian away from the stable equilibrium point (also the starting point of an episode), a reward is returned that is proportional to the 'heuristic' over energy. (as mentioned in Section 4.3.1 the heuristic is defined as the integral of the cube of $\dot{q}_1$ multiplied by its original sign). For every timestep where no flapping period is completed, the reward remains 0. In order to avoid unstable policies, a reward (punishment) of -10 is returned for every timestep where the pendulum is more than half a rotation away from the stable equilibrium; this is the out of bounds (OoB) punishment.

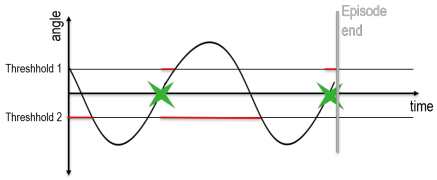
Some form of guiding punishment is useful to guide the learning algorithm to the set of policies that the ratio is defined over, thus preventing the 'plateau problem'. This section proposes two different ways of defining such a reward (Figure 4.4); both of which will be tested in the experiment.

One where a punishment (negative reward) is given at the end of an episode. This punishment is the absolute value of the distance that the trajectory missed the threshold with (set to 0.1 rad for this experiment). If the policy produces a reward during the episode no punishment is given, so the punishment should not interfere with the optimal policy in any way. (Figure 4.4a)

The second punishment design is given during the episode, in the form of a negative reward in every timestep where the threshold has not been reached yet. To avoid overly large punishments – and thus large interferences with the optimal policy – it is defined as 0.01 times the current distance from the non-reached threshold. (Figure 4.4b)



(a) Illustration of giving a punishment at the end of each unsuccessful episode. The red curly bracket indicates the magnitude of the punishment, which is only given when the episode ends.



(b) Illustration of giving a punishment during each timestep where the thresholds have not been reached. The red regions indicate the timesteps where a punishment would be given for not yet reaching that specific threshold. The green stars indicate the points where the normal reward is returned (end of one flapping period).

Figure 4.4: The two different punishment designs.

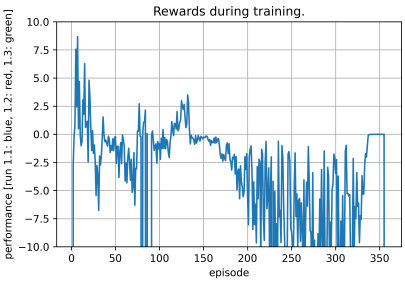
4.4 Results

Figure 4.5 shows the total reward after each training episode for the full training process. Training was done for 100,000 timesteps, with 50 timesteps per second, and between 1 and 10 seconds per episode.

Negative total rewards can be the result of: the OoB punishments, the 'guiding' punishment that is supposed to encourage periodicity of the trajecotry, or a negative numerator of the reward ratio.

Positive total rewards are the results of a positive numerator of the reward ratio, also indicating that the trajectory manages to finish a full flap at least once.

(a) Giving a punishment at the end of each unsuccessful episode. Zoomed in graphs of the rewards per episode.



(b) Giving a punishment during each timestep where the thresholds have not been reached. Zoomed in graphs of the rewards per episode.

Figure 4.5: Zoomed in graph of the rewards per episode for both punishment designs.

4.5 Discussion

None of the punishment designs discussed in Section 4.3.2 results in a stable increase of the rewards per episode. In graph 4.5b ("punishment per timestep") performance starts out well with a big peak, but degrades over time. The "punishment at the end of the episode approach" (4.5a) barely shows any positive rewards at all.

For both punishment designs, the final episode does not obtain a periodic movement producing final policy, as visible by the negative rewards in the last episodes of the performance graphs.

It is possible that an increase in training time is needed to realise an improvement in performance; specifically considering the sparsity of the reward function (even for periodic motions, rewards are only returned sporadically). Currently training takes about half an hour, so a significant increase in training steps would be possible, and would have improved the interpretability of the results. The original DDPG paper runs its experiments for 1 million steps.

All experiments in the DDPG paper [22] do show that a clear improvement should generally be visible after 100,000 time steps, which was the original motivation for capping the training time to this number. The plots in Figure 4.5 show no such improvement. In fact, plot 4.5b shows a decrease in performance over time. So it seems unlikely that the policy would improve with more timesteps.

The $50hz$ control frequency was deemed high enough to allow for a stable control law on the pendulum; specifically because of the stable nature of the pendulum dynamics. The learning performance shows some positive rewards corresponding to stable control laws, indicating that this is indeed the case. But, no experiments where run to test the influence of this control frequency. Further experiments could have provided extra insight.

The learning algorithm is tasked to simultaneously learn a control law and a periodic trajectory, while also generating desired aerodynamic effects. This combination of tasks is not often seen in successful literature, and could very well be part of the reason why performance is bad; especially when one considers that a big part of controller design is stability guarantees – rewards for which are not directly encoded in the reward function – and that the rewards are very sparse. The punishments are designed to help with this by preventing the plateau problem, reducing reward sparsity, and introducing more specific rewards to encourage periodic trajectories; but convergence is clearly still difficult.

4.6 Conclusion

This experiment implemented a DDPG agent directly on the motor torques of a simple pendulum setup, with the aim of learning a periodic motion that optimises desired effects over energy. The DDPG algorithm was not able to successfully converge to a stable policy. It could be that hyperparameter choices or lack of training time are the reason for this lack of performance. However, the experiments in the original DDPG paper show that these parameter choices are successful for solving other reinforcement learning problems.

A bigger problem seems to be the fundamental complexity of the optimisation task, in combination with the sparse rewards. Even with the punishments to guide the algorithm, the reward ratio still has to convey a lot of information about the very low level policy with very few rewards.

The next chapter of this report tries to solve these problems directly by simplifying the task that the agent has to learn. More specifically, it redesigns the task definition in such a way that the agent isn't responsible for ensuring the stability and periodicity of the policy.

5 Part 2: Fourier decomposition of the trajectory

5.1 Introduction

Part 1 of the report explored the possibility of using reinforcement learning for the optimisation of a flapping policy. Difficulties were identified with the sparsity of the ratio function in combination with reinforcement learning. Furthermore, it was speculated that the need for the policy to both act as a stable torque feedback controller, and generate periodic trajectories, all while producing the desired dynamic effects, resulted in a very difficult to optimise over task definition for the reinforcement learning agent.

This led to a reevaluation of the policy definition, with the goal to reduce the complexity of the task for the reinforcement learning agent. This new policy should: separate trajectory design from controller design, naturally encode the need for periodicity, and reduce the problem of reward sparsity.

Periodicity of the trajectory can naturally be enforced by defining the action in terms of the Fourier decomposition of the trajectory. Using this definition a single action will define the entire flapping behaviour of the agent, the agent will not act as a state feedback controller anymore. Instead, the state feedback controller will become part of the plant. The controller will be engineered by a human to be able to follow the demands of the trajectories that are generated by the learning algorithm. This action definition separates the trajectory design from the controller design in a way that naturally constrains the trajectories to be periodic. It completely removes the need for the agent to worry about stability, albeit at the cost of not being able to optimise the controller behaviour for the task. It also removes the problem of reward sparsity by defining the entire trajectory in terms of a single action; therefore ensuring the return of a reward after each action. Of course, in the process of doing this the problem is removed from the realm of reinforcement learning – where subsequent actions are able to affect each other – and into the realm of a conceptually simpler black box parameter optimisation problem.

In this chapter 3, separate experiments will be carried out: the first experiment uses the trajectory to optimise purely over the trajectory shape, no dynamics involved; the second experiment aims to test the ability of the algorithm to optimise over real setup dynamics, and does so by asking it to minimise the energy usage of the flapping policy; the third experiment actually attempts to optimise over a ratio that represent the energy efficiency of a flap. The details of experiments can be found in the method section of this chapter (5.3).

5.2 Background

5.2.1 Action definition

As mentioned, the action will be defined in terms of the Fourier series decomposition of a periodic trajectory [27, Chapter 4]:

$$x(t) = \sum_{n=-\infty}^{\infty} c_n \exp(i\frac{2n\pi t}{T}); \quad x \in \mathbb{C} \tag{5.1}$$

Here c_n represents the Fourier coefficients of the periodic function $x(t)$, T is the fundamental period of $x(t)$, and i is the imaginary unit. The Dirichlet conditions tell us that any feasible periodic trajectory could be represented in such a way.

For real valued periodic signals Equation 5.1 can be simplified to Equation:

$$x(t) = a_0 + \sum_{n=1}^{\infty} \left[a_n \cos(\frac{2n\pi t}{T}) + b_n \sin(\frac{2n\pi t}{T}) \right]; \quad x, a_n, b_n \in \mathbb{R} \tag{5.2}$$

Here a_n and b_n are real valued scalars.

A trajectory on the flapping setup is defined as a real valued periodic signal that represents the angles of each of the DoFs over time:

$$q(t) = \begin{bmatrix} q_1(t) \\ q_2(t) \end{bmatrix} \in \mathbb{R}^2; \quad \text{where: } q(t) = q(t + kT),\ k = 1, 2, ... \tag{5.3}$$

This way the full flapping trajectory on the 2 DoF setup can be approximated using:

$$q_j(t) = a_{j,0} + \sum_{n=1}^{m_j} \left[a_{j,n} \cos(\frac{2n\pi t}{T}) + b_{j,n} \sin(\frac{2n\pi t}{T}) \right] \tag{5.4}$$

Here j indicates the axis. Note how the period T is not dependent on J, this ensures that the trajectories on both axes have the same period. Each axis is approximated using a finite number of coefficients m_j.

The action is defined in terms of the fundamental frequency of the flapping motion, and the separate a_n and b_n coefficients for each axis:

$$\begin{aligned} \mathrm{a} = \begin{bmatrix} a_{1,0} & a_{1,1} & b_{1,1} & ... & a_{2,0} & a_{2,1} & b_{2,1} & ... & f \end{bmatrix}^T \\ |a_{1,n}|, |b_{1,n}| \leq max_1 \\ |a_{2,n}|, |b_{2,n}| \leq max_2 \\ f_{min} \leq f \leq f_{max} \end{aligned} \tag{5.5}$$

Each of the values in the action is bounded to a desired range.

5.2.2 The new optimisation algorithm

Using this definition, a single action defines the total flapping behaviour of the agent. Therefore the actor is not defined by a policy anymore; there is no map from state to action anymore. Instead, the actor becomes the unconnected vector of parameters defined by Equation 5.5.

The state feedback behaviour will be provided by the motor controller, which follows the trajectory defined by the Fourier coefficients which are provided by the action, see Figure 5.1.

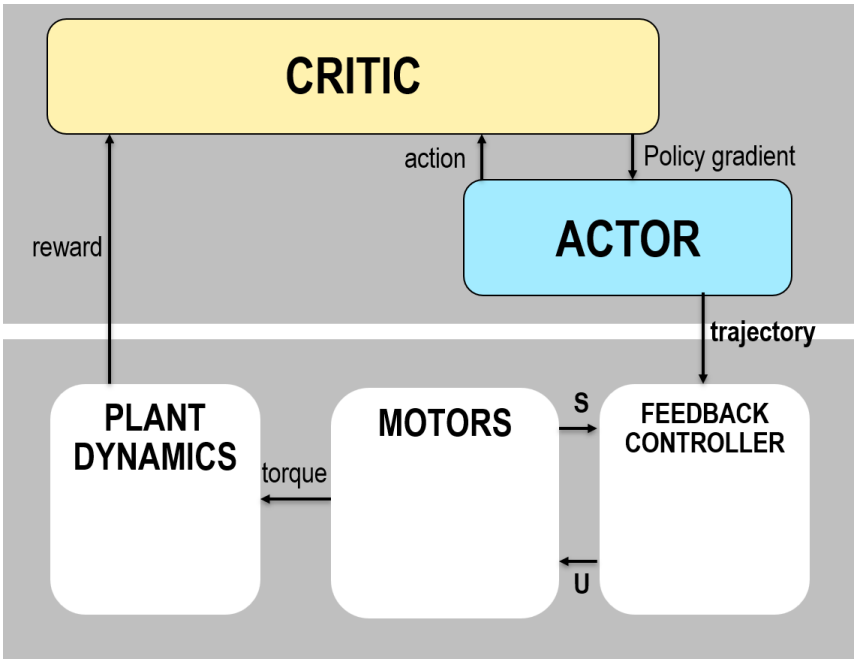


Figure 5.1: New learning scheme.

Changing the task definition ins this way fundamentally changes the definition of the actor and the critic.

The actor used to be a map from a state to a reward. Because a single action now defines the whole trajectory – and thus the whole flapping behaviour – the idea of a state is not relevant anymore. So the definition of the actor is changed from a map from state to action, to a vector of parameters that define the action and therefore also the trajectory (Equation 5.5). The actor is still updated using a gradient, which will still be estimated using the critic.

The critic used to be an estimate of the map between state and action, and expected future reward. It has now become a map from action to reward:

$$\begin{aligned} \text{Former critic:} \quad & C : s_t, a_t \rightarrow J_t; \\ \text{Current critic:} \quad & C : a \rightarrow r \end{aligned} \tag{5.6}$$

Symbols s, a, and r represent the state, action, and reward respectively, and J represents the future expected reward.

The reason for this change is the fact that a single action now defines the full flapping behaviour. So the idea of a state that influences the choice of action becomes irrelevant. For the same reason, the idea of a future expected reward becomes irrelevant; there is no sequence of states, actions, and rewards that can influence each other. So our critic stops having to estimate a total future expected reward, and can instead predict a single reward given an action.

Therefore the critic network update targets do not have to be determined by the Bellman equation anymore. So the received reward can be used directly as a target to train the critic network. Due to the lack of future action choices affecting the results from the current action choice, the learning process of the Q network becomes stationary, and theoretically much more stable.

The critic is still used to estimate the gradient that is used to update the actor. But the gradient calculation can be greatly simplified:

$$\nabla_a r \approx \mathbb{E}\left[\nabla_{\theta^\mu} Q(a|\theta^\mu) \nabla_{\theta^a} a\right] \tag{5.7}$$

In this equation θ^μ represents the parameter vector of the Q network (neural network representing the critic), and θ^a represents the parameters of the action.

The final algorithm becomes a much simplified version of the actor-critic algorithm used before (DDPG [22]). The Q function is still represented by a neural network of the same basic design as in [22], but with the extra input for the state removed. The actor is represented by a single layer neural network with a constant input. This way, the build in Adam implementation of Pytorch [2] can be used to optimise the parameters. The outputs of the actor network are filtered through a tanh layer and multiplied by the maximum values for each coefficient, as defined in Equation 5.5. Pseudocode for the lagorithm can be found in Algorithm 1.

Algorithm 1 Actor-critic optimisation algorithm for Fourier coefficients.

Randomly initialise critic network $Q(a|\theta^Q)$
Randomly initialise actor network $\mu(s|\theta^\mu)$.
initialise replay buffer R
while $episode \leq max\_episodes$ **do**
  select action with noise: $a = \mu(s|\theta^\mu) + \mathbb{N}$
  execute action and observe reward: r
  store the state action and reward (s, a, r) in R

  randomly select training batch B from R
  Update $Q(a|\theta^Q)$ by minimising MSE between its prediction and the r_B
  Update $\mu(a|\theta^\mu)$ using $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i^N [\nabla_a Q(a|\theta^Q)|_{a=\mu(1|\theta^\mu)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=1}]$
end while

5.2.3 Fundamental limitations due to controller design

The research in [7], [11], And [13] discus the effects of controller design on plant dynamics and the cancelling effect feedback control has on compliance in a system. They suggest a heavy reliance of biological systems on feed-forward control to maintain the desired compliance of their actuators. For example, the ability of a human to carefully and precisely lift a heavy object.

It is suggested that this feed-forward behaviour is constantly adjusted to the environment and to the task. Intuitively, one would expect a bird to do the same during flight: constantly adjusting its control strategy to the current environmental conditions to achieve optimal performance. This suggests that controller design, including the balance between feedback and feed-forward, is an important part of optimising a flapping policy. In the current learning structure the actor is not able to adjust the controller; this limits the optimisation to the given controller which is in no way guaranteed to be optimal for flapping flight. The flexibility of the wing will of course not be altered by the controller design. Therefore the optimal policy should still be able to take advantage of some of the effects caused by flexibility.

5.3 Method

This section outlines the method of three separate experiments: the first experiment uses the trajectory to optimise purely over the trajectory shape, no dynamics involved (5.3.1); the second experiment aims to test the ability of the algorithm to optimise over real setup dynamics, and does so by asking it to minimise the energy usage of the flapping policy (5.3.2); the third experiment actually attempts to optimise over a ratio that represent the energy efficiency of a flap (5.3.3).

Unfortunately, the real setup was not yet usable at the time of the experiments; therefore a placeholder setup had to be constructed (Section 3.2). This placeholder setup has no dedicated force sensor, so the data that is available for evaluation is limited to the control value of the build in position controller.

In addition, a simplified digital model of the placeholder setup was made. This model has the same geometry, and in- and outputs as the real setup; but with a simple aerodynamics model (Section 3.3).

5.3.1 Experiment 1: Fitting the action to a trajectory shape

This experiment aims to test the ability of the actor critic architecture to optimise the parameters of the action to approximate a known periodic shape; specifically a square wave is used as the target function. This is the first test to test the validity of the algorithm discussed in Section 5.2.2.

The optimisation algorithm is as described in section 5.2.2. The trajectory that is generated by the policy is optimised directly over the shape of the goal trajectory; so no setup dynamics are taken into account. The algorithm is given full control over all parameters of the action, except for the fundamental frequency: separate experiments are done with and without the fundamental frequency being a changeable parameter. The maximum values of each parameter can be found in Table 5.1.

Parameter	**Value**	**Parameter**	**Value**
Learning rate critic	1e-2	Layer sizes Q	300, 500, 300
Learning rate policy	5e-3	amplitude limits coefficients	85 degrees
Batch size	512	amplitude limit DC coefficient	85 degrees
Buffer Size	1,500	number of coefficients	5
number of episodes	1,500		
number of random policies	500		

Table 5.1: Hyperparameter values of the optimisation algorithm.

The reward function is defined as the energy of the difference between the square wave signal and the current trajectory (Equation 5.8), which should always be minimised by the correct Fourier coefficients of a signal, given the set of orthogonal basis functions [27, Chaper 3.2]. As the basis functions of a Fourier decomposition of a periodic trajectory wouldn't be defined without the fundamental frequency, this frequency is set to be a constant value instead of being chosen by the learning agent; however, since we are interested in evaluating the algorithm's ability to optimise over all parameters of the action, additional experiments are done with the fundamental frequency as a changeable action parameter.

$$\text{reward} = -\int_{<T_S>} |S(t) - P(t)|^2 dt \tag{5.8}$$

In Equation 5.8 $S(t)$ represents the square wave, $P(t)$ the trajectory determined by the policy, and the symbol $< T_S >$ indicates that the integral is over a single period of the square wave (the region we want to approximate).

5.3.2 Experiment 2: Minimising the energy use on a physical setup

This section aims to explore the feasibility the Fourier parameter based optimisation approach by optimising over a reward function that is derived from setup dynamics. Tests will be done on the real placeholder setup and on the simulated placeholder setup.

The reward function

The goal of these experiments is to test the ability of the algorithm to optimise over a reward function that's defined on the sensor data of a real setup; that way, the rewards will be influenced by the full setup dynamics, including but not limited to: the trajectory determined by the actor, the controller dynamics, the physical behaviour of the setup, sensor noise etc.

Because these experiments are meant to be a proof of concept of training the policy on the real setup – and not a full solution to the optimisation problem – the reward function is defined in a uncomplicated way: the negative value of the total energy consumed by the system during one period.

Sensor data on the available setup is limited to the desired torque output of the controller on axis $q1$; therefore, no direct energy measurement is available on the real setup, see discussion in section 3.2.3. Instead power consumption is estimated using the absolute value of the aforementioned control value. This value is then numerically integrated using a simple Riemann sum to get an very rough estimate of the total energy usage over one period.

This energy estimate does not aim to be an accurate estimate of the real energy usage of the setup, although it does seem reasonable to suggest that it is in some way proportional to the real energy usage. Instead it represents a simple reward function, that is based on real setup dynamics, and results in a predictable optimal policy: a policy with as little movement as possible. In other words, a policy with it's amplitudes approaching 0 as training progresses.

The learning scheme

For this experiment the simplified actor-critic algorithm discussed in Section 5.2.2 is used.

The DC coefficient of axis $q1$ is set to zero to force the trajectory to always centre around the stable equilibrium, the other limits are set to be reasonable with respect to the physiscal limits of the setup, see Table 5.3.

The size, hyperparameters, and optimiser of the algorithm are based on the recommendations in the DDPG paper [22]. Since each timestep requires the setup to perform en entire flapping episode, the time needed per timestep is quite large; therefore the network is only trained for 1300 episodes. This takes about 4 to 5 hours of training time on the real setup, which most of which is taken up by the execution time of the policy. Which is a convenient amount of time, as it takes up a single morning/afternoon (specifically because the setup had to be supervised during execution for safety reasons). The learning rates and batch size are increased to accommodate the decrease in the number of steps. This is possible due to the simplified nature of the problem compared to full reinforcement learning.

Lastly, the training process is started with five hundred randomly generated actions. These actions are used for the initial exploration and training of the critic network. Preliminary tests have shown it improves convergence on this problem.

Parameter	**Value**	**Parameter**	**Value**
Learning rate critic	1e-2	Layer sizes Q	400, 300, 300
Learning rate policy	5e-3	amplitude limits coefficients q_1	90 degrees
Batch size	1,000	amplitude limits coefficients q_2	20 degrees
Buffer Size	5,000	amplitude limit DC coefficient q_1	0 degrees
number of episodes	1,300	amplitude limit DC coefficient q_2	20 degrees
number of random policies	500	number of coefficients q_1	7
minimum punishment (simulation/ real)	70/ 70	number of coefficients q_2	7
punishment scalar (simulation/ real)	4/ 10		
reward scalar (simulation/ real)	5/ 1		
noise scalar (multiplied by output limits)	0.3		

Table 5.2: Hyperparameter values of the optimisation algorithm.

As mentioned, each timestep is now defined as a full episode. Here an episode should be at least a full flapping period to allow to proper calculation of the reward ratio. To prevent the inclusion of transient effects, both the setup and the simulation are run with half a period head start before storing reward data; thus requiring 1.5 period to be executed per episode.

Out of bounds punishments

To protect the setup from physical harm, there is a limit to the maximum amplitude that the trajectory can take. This limit is set to 180 degrees away from the equilibrium for q_1 and 90 degrees away from the 0 degree point of q_2. (0 degrees on q_2 is defined as the point in between a fully opened and a fully closed wing angle).

Punishments are given for Out of Bounds (OoB) trajectories; furthermore, OoB trajectories are not executed and evaluated to prevent damage to the setup. The punishment consists of a proportional and constant part. For the proportional part (Figure 5.2), the number of degrees that a trajectory is out of bounds is multiplied by the punishment scalar and subtracted from the reward (which is always 0 since the trajectory won't be executed if a boundary is crossed). This makes the punishments proportional to the violation, providing a gradient for the learning algorithm to follow.

To avoid the creation of new minima at the border between valid and invalid trajectories, a constant term is added to the punishment. More specifically, in this experiment the theoretical maximum reward is 0 for a trajectory with no energy consumption. Additionally, if no constant punishment was added to the proportional part, the theoretical minimum punishment would be arbitrarily close to 0 (for a arbitrarily small violation of the setup limits); in other words arbitrarily close to the theoretical optimal reward. This could result in the optimisation algorithm moving towards the "wrong" optimum; namely one created by the OoB punishment design and not by the value function we are actually interested in.

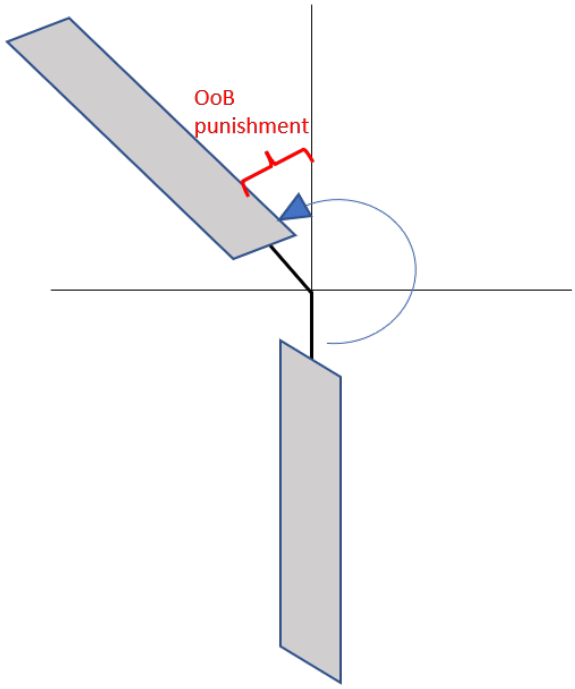


Figure 5.2: Illustration of the proportional element of the OoB punishments on q_1.

5.3.3 Experiment 3: Optimising over a ratio

Experiment tests the capability of the algorithm to optimise over actual setup dynamics using a simple reward function. This section aims to explore the optimisation of the parameters over a ratio; where the denominator represents some energy measurement and the numerator represents a goal or heuristic that we want to maximise as energy-efficiently as possible.

In this experiment multiple runs will be done: first with similar maximum Fourier coefficient values as in the energy minimisation experiment (I), and a second set of runs with the maximum values reduced (II). The purpose of this second set of experiments is to evaluate the learning performance without running out of bounds, thus avoiding the punishments.

No experiments are done on the physical setup due to unsatisfactory results when training on the digital setup (see coming sections); instead it was opted to do more experiments in simulation with the goal to create more insight into the lack of performance.

The reward function and out of bounds punishments

Just as in the energy minimisation experiment, the reward function design is severely limited by the lack of sensor data available on the setup. The numerator of the reward function ratio will be defined in terms of a integral over the controller output of axis q_1; thus the numerator will be directional, promoting torque output in one direction and discouraging it in the other. The aim of this is to promote a non-symmetrical flapping motion.

The denominator will be defined in terms of an energy estimate. In this case, the square of the controller output was used. This makes for an non-directional value that scales differently to the numerator, and should theoretically scale up as energy usage increases.

$$reward = \frac{\int_T u \cdot dt}{\int_T u^2 \cdot dt} \tag{5.9}$$

Equation 5.9 shows the reward function for this experiment. Here, u represents the controller output of q_1 and T represents one period of the flapping trajectory.

Punishments given for out of bound (OoB) trajectories are defined in the same way as in the energy minimisation experiment (Section 5.3.2): a combination of a linear part to provide a gradient for the algorithm to follow, and a constant part to prevent the formation of local optima. Unlike in the energy minimisation experiment, the maximum reward in this experiment can exceed 0. The optimum policy, and its accompanying maximum reward, are unknown.

Hyperparameter values

Hyperparameter values are largely the same as in the energy minimisation experiment (Section 5.3.2). The reward value scalar is set to 3 in order to increase the rewards size to a reasonable range (between -1 and 1). This range is smaller than in the energy minimisation experiment. To compensate for the smaller magnitude of the policy gradient as a result, the policy learning rate is increased.

The number of coefficients per axis is decreased from 7 to 4. This creates a lower dimensional parameter space to optimise over.

Parameter	**Value**	**Parameter**	**Value**
Learning rate critic	1e-2	Layer sizes Q	400, 300, 300
Learning rate policy	1.5e-1	amplitude limits coefficients q_1 (I/II)	90/40 degrees
Batch size	1,000	amplitude limits coefficients q_2 (I/II)	20/40 degrees
Buffer Size	6,000	amplitude limit DC coefficient q_1	0 degrees
number of episodes	6,000	amplitude limit DC coefficient q_2	20 degrees
number of random policies	500	maximum frequency	1hz
minimum punishment (simulation)	1	minimum frequency	0.25hz
punishment scalar (simulation/ real)	0.0025	number of coefficients q_1	4
reward scalar (simulation/ real)	3	number of coefficients q_2	4
noise scalar (multiplied by output limits)	0.3		

Table 5.3: Hyperparameter values of the optimisation algorithm.

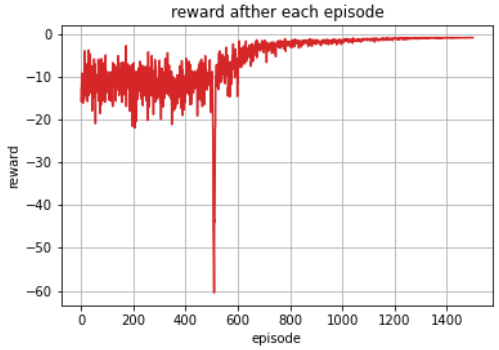
5.4 Results

5.4.1 Fitting the action to a trajectory shape

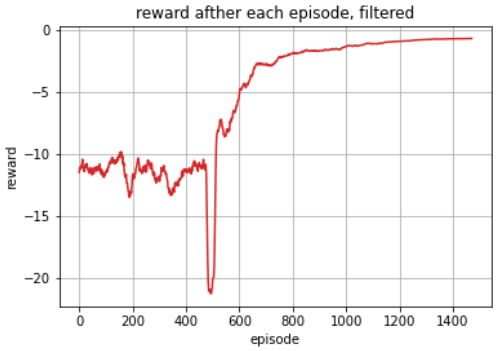
This section shows the results of the experiments that aim to fit the trajectory to a square wave. The first part shows the results of the experiment with a constant fundamental frequency; the second part shows the results with the fundamental frequency as part of the action.

Results with a constant fundamental frequency

Figure 5.3 shows the rewards during training for the experiment with a constant (non-changeable by the agent) fundamental frequency. Figure 5.3a shows the original data, and Figure 5.3b shows the data filtered using a low pass filter to uncover a general trend line.



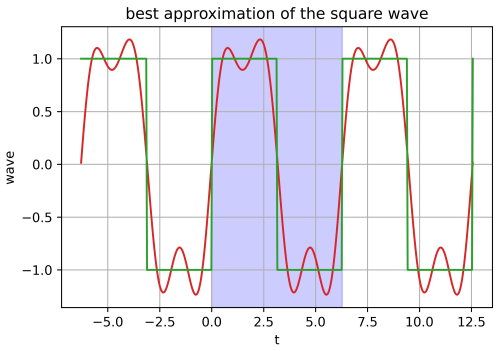
(a) Rewards during training.



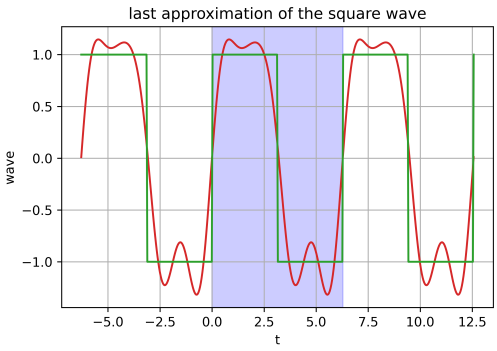
(b) Rewards during training; filtered using uniform convolution of 30 samples wide.

Figure 5.3: The rewards during the training episode.

Figure 5.4 shows the trajectory generated by the learning algorithm in red, and the square wave it is trying to approximate in green. The blue area indicates the exact region over which the reward in Equation 5.8 is calculated. Figure 5.4a shows the best trajectory that was achieved during the entire training session, Figure 5.4b shows the last trajectory achieved during training.



(a) Best action.



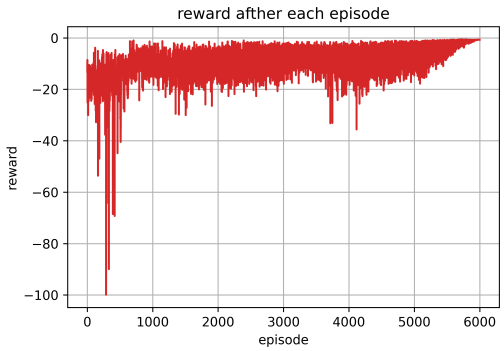
(b) Last action.

Figure 5.4: The best and last action compared to the square wave. Reward is calculated for $0 \leq t \leq 2\pi$ (blue area).

Results with a changeable fundamental frequency

Evaluation over a single period:

Figure 5.5 shows the rewards during training for the experiment with a non-constant fundamental frequency (the agent determines the frequency of the trajectory). Figure 5.5a shows the original data, and Figure 5.5b shows the data filtered using a low pass filter to uncover a general trend line. Simulation time was increased of 6000 to get the reasonable performance.

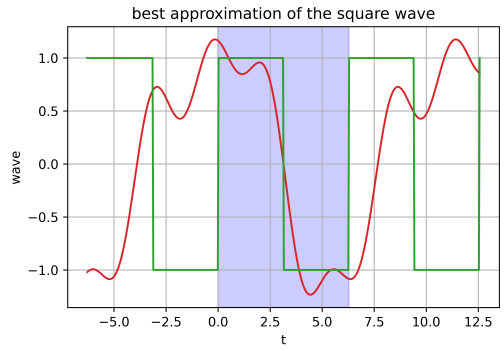


(a) Rewards during training.

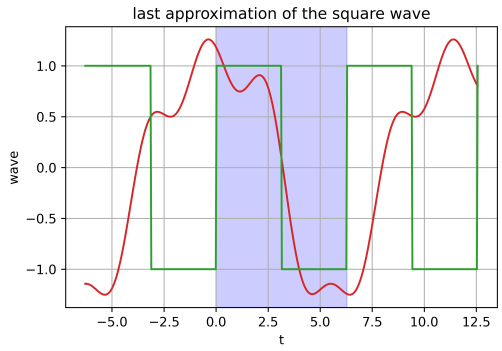
(b) Rewards during training; filtered using uniform convolution of 30 samples wide.

Figure 5.5: The rewards during the training episode. The fundamental frequency is set by the actor. The reward is calculated for $0 \leq t \leq 2\pi$.

Figure 5.6 shows the trajectory generated by the learning algorithm in red, and the square wave it is trying to approximate in green. The blue area indicates the exact region over which the reward in Equation 5.8 is calculated. Figure 5.4a shows the best trajectory that was achieved during the entire training session, Figure 5.4b shows the last trajectory achieved during training. The approximation is close within the region that is used to calculate the reward, but inaccurate outside. The fundamental frequency does not match the frequency of the square wave at all.



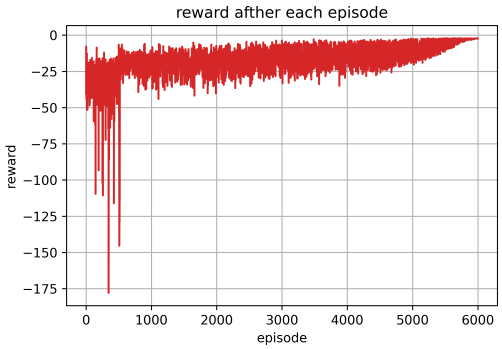
(a) Best action.



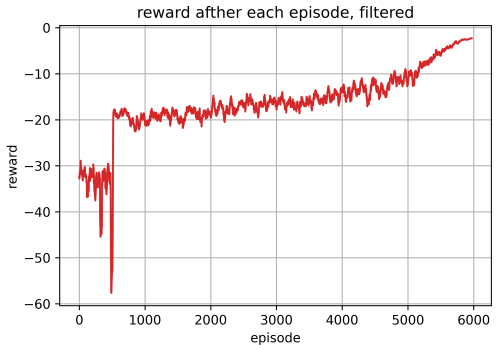
(b) Last action.

Figure 5.6: The best and last action compared to the square wave. The fundamental frequency is set by the actor. The reward is calculated for $0 \leq t \leq 2\pi$ (blue area).

Evaluation over two periods:
Figure 5.7 shows the rewards during training for the experiment with a non-constant fundamental frequency (the agent determines the frequency of the trajectory), but with a larger region over which the reward is calculated (2 periods: $-2\pi \leq t \leq 2\pi$) to encourage correct periodicity of the action. Figure 5.7a shows the original data, and Figure 5.7b shows the data filtered using a low pass filter to uncover a general trend line.



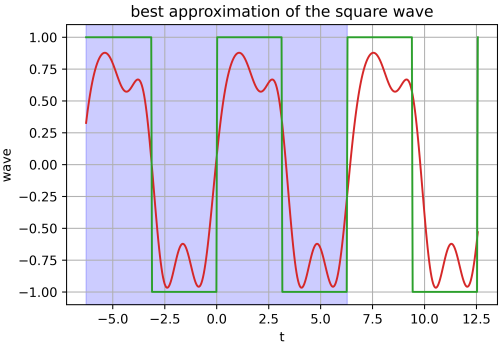
(a) Rewards during training.



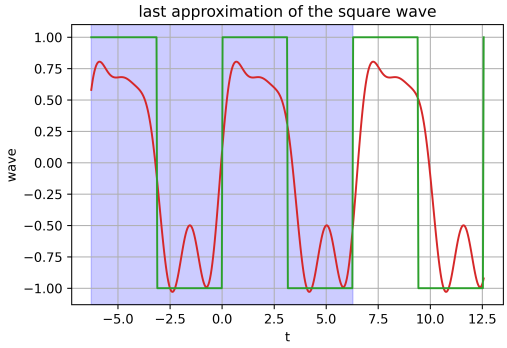
(b) Rewards during training; filtered using uniform convolution of 30 samples wide.

Figure 5.7: The rewards during the training episode. The fundamental frequency is set by the actor. The reward is calculated for $-2\pi \leq t \leq 2\pi$.

Figure 5.8 shows the trajectory generated by the learning algorithm in red, and the square wave it is trying to approximate in green. The blue area indicates the exact region over which the reward in Equation 5.8 is calculated. Figure 5.8a shows the best trajectory that was achieved during the entire training session, Figure 5.8b shows the last trajectory achieved during training.



(a) Best action.

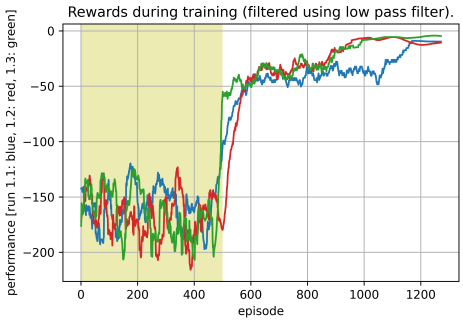


(b) Last action.

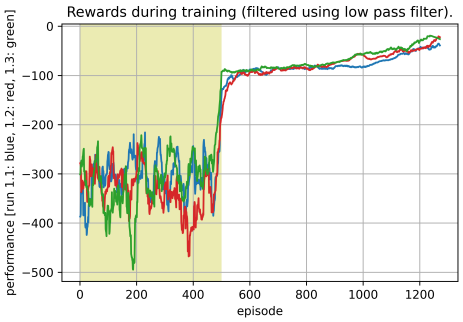
Figure 5.8: The best and last action compared to the square wave. The fundamental frequency is set by the actor. The reward is calculated for $-2\pi \leq t \leq 2\pi$ (blue area).

5.4.2 Minimising the energy use on a physical setup

Running the optimisation algorithm three times results in the reward function progression seen in Figure 5.9. There is a difference between the magnitude of the reward values in simulation and on the real setup. This was partly fixed by using a constant reward scalar.



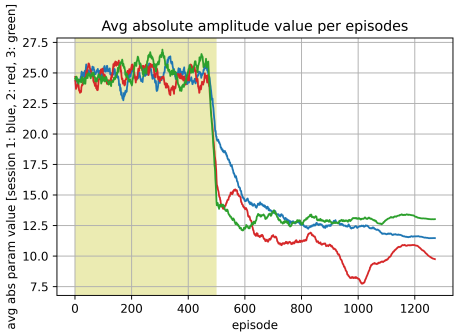
(a) Digital setup.



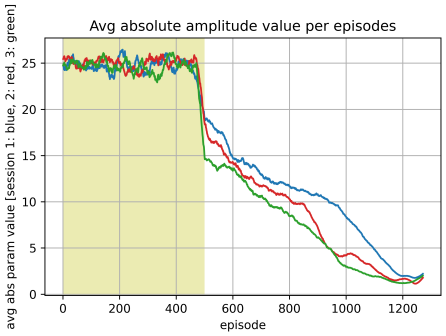
(b) Real setup.

Figure 5.9: Reward function during training on both real and simulated setup – 3 runs each. Yellow area is where the policy is completely randomised. (Filtered using uniform convolution of 30 samples wide to improve visibility, original data in Appendix C).

Intuitively, the optimal policy for this experiment minimises amplitudes of the Fourier parameters. Figure 5.10 shows the average value of the amplitude of the Fourier parameters over time.



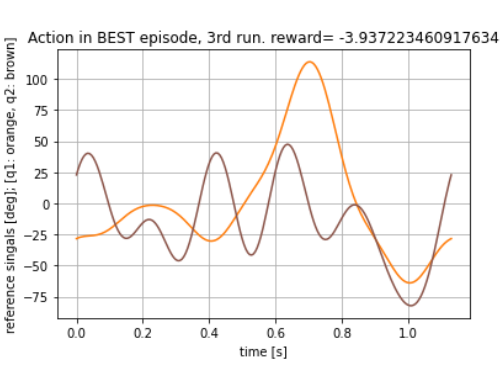
(a) Digital setup.



(b) Real setup.

Figure 5.10: Average of the absolute value of the amplitudes of each episodes action during training on both real and simulated setup – 3 runs each. Yellow area is where the policy is completely randomised. (Filtered using uniform convolution of 30 samples wide to improve visibility, original data in Appendix C).

Figure 5.11 shows the best performing and last trajectory obtained during run 3. Here the resulting trajectory is shown from adding up the Fourier coefficients determined by the action. This trajectory given as reference signal the controllers on setup, resulting in its execution. The orange line indicates the angle of q_1 (the up and down flapping angle), while the brown line indicates $q2$ (the wing angle).



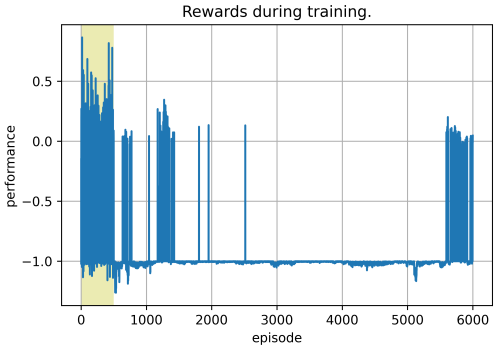
(a) Digital setup.

(b) Real setup.

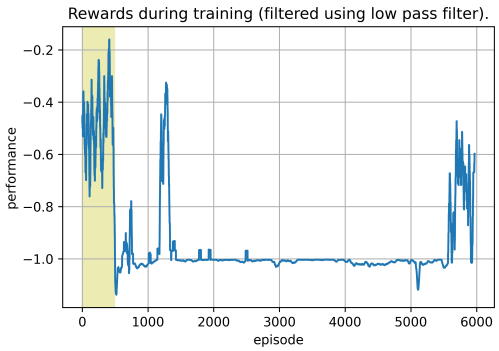
Figure 5.11: Reference signals resulting from the actions of the best episodes of the 3rd training run with both the simulated and real setup.

5.4.3 Optimising over a ratio

Figure 5.12 shows the runs with the large maximum Fourier parameters (I).



(a) Rewards during training.

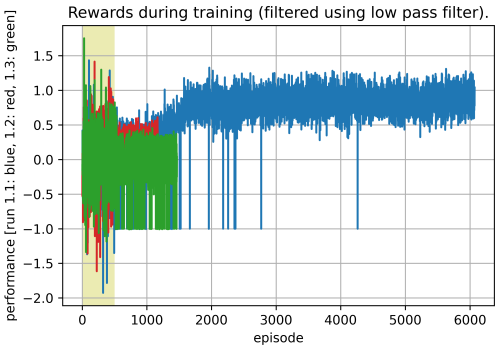


(b) rewards during training, data filtered with low-pass (uniform convolution of 30 samples wide).

Figure 5.12: The rewards during the training episode, simulated plant. large Fourier coefficient maximum (I). Yellow area indicates where the policy is completely randomised.

Figure 5.13 shows three runs with smaller Fourier coefficient maximums, making it less likely for the action to be out of bounds. Multiple runs where done to get a better general idea of the performance of the algorithm. 2 of the runs where stopped prematurely due to a complete disappearance of both the Q network gradients and the policy gradients.

Performance of run two and three (red and green) shows almost no improvement after the first 500 random policies and all the network gradients disappear. The first run (blue) does show some nice improvement only for performance to stagnate and the gradients to disappear as well. The highest performance in each training run is achieved during the first 500 episodes which use completely random policies for exploration purposes, so algorithm performance is worse than random.

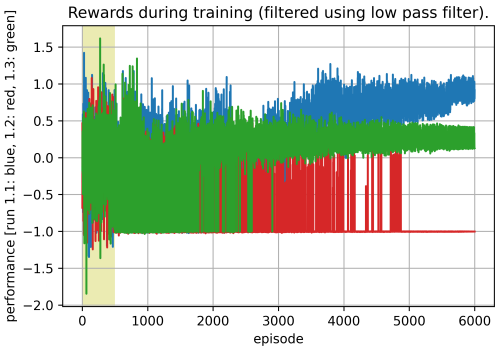


(a) Rewards during training.

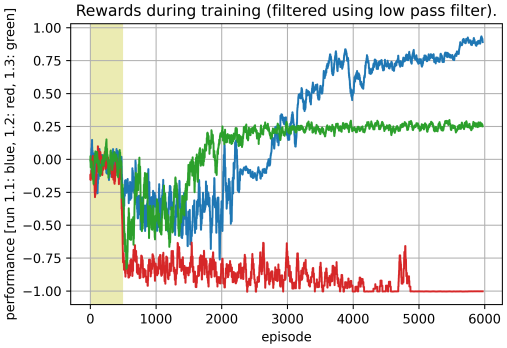
(b) rewards during training, data filter with low-pass.

Figure 5.13: The rewards during the training episode, simulated plant. Small coefficient maximum (II). Run 1.2 and 1.3 where cut short after the policy gradient disappeared. Yellow area indicates where the policy is completely randomised.

In order to gain a better understanding of algorithm performance, multiple additional optimisation runs where performed with alternative hyperparameter choices: runs with a lower exploration noise levels, runs with higher noise levels, runs with a lower policy learning rate, and runs with a smaller batch size and higher noise. Most of these runs perform similar or worse to the runs with the original hyperparameters. The results of these runs can be found in Appendix A. The runs with higher exploration noise are an exception to this as they performed better than the original experiment. These runs use maximum noise values of 0.8 times the maximum parameter values; the rest of the hyperparameters are the same as in the runs from Figure 5.13. As in previous experiments, the scalar of the noise values is decreased linearly during training, until the noise reaches a value of 0 in the last episode. The performance of these runs can be found in Figure 5.14.



(a) Unfiltered rewards during training.



(b) reward data filtered with low-pass (uniform convolution of 30 samples).

Figure 5.14: The rewards during training for all three runs with higher noise values.

Two of the three runs quickly get stuck during training: run 2 gets stuck at the minimum punishment; Run 3 gets stuck around a reward of 0.25. Both of these loose their policy and critic gradients quickly after getting stuck.

Run 1 shows steady improvement during training but does not ever reach the performance that is achieved by chance during the 500 random policies at the start of the training session. The policy gradient of this run has not disappeared at the end of training.

5.5 Discussion

5.5.1 Fitting to a trajectory shape

The aim of this experiment is to test the ability of the actor critic architecture to optimise the parameters of the action to approximate a known periodic shape.

With the fundamental frequency of the action set to match the fundamental frequency of the square wave, performance of the algorithm is very stable. The approximation fits the square-wave well considering the number of basis coefficients used. Not much time was spend optimising the parameters, so it is possible that faster convergence can be achieved

With the fundamental frequency as part of the action, convergence becomes much slower; however, the algorithm is still able to optimise the function. When the reward is calculated over a single period of the square wave, this results in a good approximation in that period, and bad approximation in the following periods. Evaluating the action over two periods of the squarewave fixes this problem; but the final approximation still isn't as good as with a fixed fundamental frequency. finding the correct fundamental frequency made the task quite a bit more difficult to optimise. It is not clear if this is generally true, or just a characteristic of this optimisation problem.

5.5.2 Minimising the energy use on a physical setup

Figure 5.9 shows the algorithm moving nicely towards the highest possible reward after completing the 500 random policies at the start of each training session. Interestingly, the simulated setup does not perform as cleanly as the real setup. This could indicate a more difficult to traverse reward function in simulation versus the real setup, likely due to modelling errors.

Also interesting to note that maximum amplitude of the last action in simulation was quite high (Figure 5.11a) even though the reward function design was expected to encourage the amplitude of the trajectory to go to 0. Nonetheless, Figure 5.9a shows a nice improvement during training. Indicating that the algorithm did move toward some sort of (local) minimum, even if it is not in the way that was expected. It is possible that the single large peak in the trajectory of 5.11a is quite energy efficient under the simulated physics.

Overall, the reward functions do show a pretty stable improvement in rewards. This shows that the optimisation algorithm works for simple problems, and that the algorithm can learn from data collected on a real setup.

5.5.3 Optimising over a ratio

Figure 5.12 only shows a single run of the learning algorithm. This run was done in simulation only because slow convergence caused the need for many training steps, which wasn't feasable on the real setup. In addition, because of unsatisfactory performance more time was spend on additional hyperparameter tuning (See Appendix A). The highest performance is achieved during the first 500 episodes which use completely random policies for exploration purposes.

Afterwards, the algorithm spends most of its time stuck at the minimum reward; It seems to have difficulties navigating the reward function, it is not able to consistently increase the reward from the minimum punishment.

The runs with smaller maximum Fourier coefficient values (Figure 5.13) tend to get stuck and loose their gradients. The cause for this could be many: There could be problems with the reward function design, including the OoB punishments; There could also be problems with Q function convergence and its ability to estimate the gradient. This could be caused by the hyperparameter choices or the network design, however the network is very similar to that in the DDPG paper, where theoretically more complicated reinforcement learning problems where solved, so it is less likely that this is the problem; Lastly, the number of training steps is

not very large – although still four times as large as the energy minimisation experiment – so it is possible that an improvement in performance could be seen after even more steps. The reason for not increasing learning time is that, based on the energy minimisation experiment, learning time on the real setup would already take a full 24 hours.

As the gradients disappear it seems unlikely that more training time would improve performance. Hyper-parameter choices could be part of the reason for bad performance; therefore, some more runs were performed with: less exploration noise, more noise, a smaller learning rate, and a combination of a smaller batch size and a higher noise level. These runs can be found in appendix A, except for the run with higher exploration noise which can be found in Figure 5.14. Higher noise levels improve performance a bit (0.9 final reward), but again performs worse than the best of the randomly picked policies at the start of the training sessions (1.4, 1.1, and 1.6 respectively for run 1, 2 and 3). Higher exploration noise also seems to prevent the disappearance of the policy gradient.

An interesting note about the runs with higher noise levels (Figure 5.14), is that the second run (red) converges to a value of -1, which is the least severe OoB punishment; apparently, the maximum Fourier parameter values were not set quite small enough to completely prevent going OoB. It converges to this value from a initial higher reward value and then stays there for the rest of training. It is not quite clear why the algorithm has a tendency to converge to this reward value; if the cause was a local minimum one would not expect the algorithm to move there from a higher initial reward.

Lastly it is good to note the potential variability of the magnitude and the gradient of any ratio function; potentially making things difficult for a gradient descent algorithm in the form of local optima or significant changes in gradient magnitude.

5.6 Conclusion

5.6.1 Fitting to a trajectory shape

The experiments show that the actor-critic architecture can be used to optimise the action parameters over a black box reward function. Including the fundamental frequency in the action makes optimisation more difficult, but it is not clear whether this is a characteristic of the specific experiment, or of the optimisation algorithm and action definition in a more general sense. The sample efficiency of the algorithm is not very high.

5.6.2 Minimising the energy use on a physical setup

This experiment shows that the algorithm can be used to optimise the behaviour of the real setup, albeit with a simple task definition.

5.6.3 Optimising over a ratio

The goal of this experiment was to test the algorithms ability to optimise over a reward function defined in term of a ratio where the denominator represents energy use, and the numerator represents a heuristic designed to lead to asymmetrical flapping behaviour.

This experiment does not lead to great results: the algorithm has a tendency to get stuck in the transition region between valid and invalid trajectories. When the OoB punishments are avoided the algorithm shows some convergence, but performance is worse than just taking random guesses.

6 General discussion

The first section of the report (Chapter 4) explores the first sub question of the report: "Can we use continuous reinforcement learning techniques to directly learn a state feedback policy that the solves optimisation problem?". This is done through the direct implementation of the DDPG algorithm on the motor torques of a simplified setup model (pendulum).

In the theoretical background of the experiment various difficulties are identified with this optimisation problem. These are the sparsity of the reward and the risk of the plateau problem, and the complexity of the task definition (stability, periodicity, and aerodynamics).

The experimental results show that these difficulties are indeed a problem, and that designing a additional reward to guide the algorithm is not straight forward (the punishment). More experiments could have been done to explore the influence of parameter choices and learning time.

From these results, the report directly moves towards an alternative action definition that takes us away from the reinforcement learning problem and removes many of the aforementioned difficulties. Alternative ways to solve the previously identified problems are not discussed (see future works for a suggestion). So the chapter answers the question in so far that uncovers some significant difficulties with applying reinforcement learning to the setup, and suggests that moving away from the reinforcement learning framework is desirable for our optimisation problem. But it does not manage to exhaustively answer the question whether reinforcement learning could be used to successfully learn a state feedback policy.

The second section of the report (Chapter 5) tries to answer the second sub question of the report: "Can we repurpose the actor critic architecture from reinforcement learning to optimise over Fourier coefficients representing a flapping trajectory?". It does this in the way of various experiments designed to test the validity of the new optimisation method.

Experimental results show the ability of this architecture to optimise the trajectory shape to a predetermined shape, and the ability to optimise a simple reward function over setup dynamics; albeit in with less sample efficiency than hoped.

Optimisation over a ratio between a 'desired effect' and a energy estimate does not go well, with algorithm performance being worse than a equivalent number of random guesses, and with the OoB punishments stopping learning. In addition, even without the OoB punishments the algorithm often gets stuck during learning.

Performance of the algorithm might be different for different ratios, specifically different definitions of the numerator value. The current value does not directly represent interesting dynamic effects because it was designed to work on the placeholder setup.

In a more general sense, the use of gradient based optimisation methods for a reward function defined in terms of a ratio could be problematic due to the possibility of large magnitude changes of the ratio. It is possible that this is part of what causes the algorithm to stop improving after a set number of training steps. Of course the actor critic algorithm architecture does not have to be used to optimise the Fourier coefficient values, and the need to train a full neural network to estimate the gradient used for optimisation is not very efficient.

Therefore, the use of the actor critic architecture to optimise the Fourier coefficients over a ratio does not seem especially effective. But that does not take away the benefits of this action definition in terms of how it simplifies the optimisation problem as compared to reinforcement learning, and the guarantees it gives on periodicity and stability.

7 General conclusion

In an attempt to optimise the energy efficiency of a flapping policy, various techniques where discussed: numerical optimal control techniques such as shooting and direct collocation techniques; reinforcement learning techniques, specifically DDPG; and a simplification of the actor critic method where the policy is defined in terms of Fourier coefficients describing a trajectory.

All of these techniques where discussed with the concept of a ratio between a desired effects and energy as a reward function. This reward function gives a direct representation of the energy efficiency of the flapping. This reward function design does however present challenges in the form of reward existence/sparsity and variability of the reward magnitude.

Potential difficulties where encountered regarding using reinforcement learning to directly learn a joint-torque state-feedback law on the flapping setup; specifically, the potential non-existence of the chosen reward function, in combination with the sparsity of the reward function for scenarios where it does exist, makes the problem diffic ult. It was attempted to solve the non-existence of the reward by adding a "punishment" function, but this still did not lead to good convergence of the problem.

This lack of performance was theorised to stem from the complexity of the task that the reinforcement learning agent is asked to learn; namely the combination of controller stability, periodic trajectory generation, and the generation of the desired effects. Which is a combination of sub-tasks not often seen in successful literature, and of which the reward function offers very little guidance, even with the additional punishments added to the reward.

Simplifying the task definition using Fourier decomposition of the flapping trajectory solves most of these problems. The actor critic architecture can be used to optimise the action parameter vector over a black box reward function. This is shown in the trajectory shape fitting experiment and the energy minimisation experiment. The algorithm also works in combination with the real setup (again the energy minimisation experiment). However, it does not seem to be very sample efficient in its optimisation.

Optimising over the ratio function – where the denominator represents energy use, and the numerator represents a heuristic designed to lead to asymmetrical flapping behaviour – does not lead to great results. The algorithm has a tendency to get stuck in the transition region between valid and invalid trajectories. When the OoB punishments are avoided the algorithm shows some convergence, but performance is worse than just taking random guesses. It is possible that another black box parameter optimisation technique can achieve better performance.

8 Future work

This report discusses two possible techniques for optimising the flapping behaviour of the wind-tunnel setup, each represented by one of the sub-research questions. This section on possible future work will be split along the same lines.

The first technique that was discussed is reinforcement learning for optimisation, with the agent directly controlling the motor torques. This task could be simplified by separating controller and trajectory design just like the Fourier decomposition technique. For example by making the agent output trajectory waypoints or velocities instead of motor torques, and having those executed using a human designed controller. Periodicity of the trajectory can be encouraged by using a human designed state machine in the policy network [17], which has in the past been shown to be successful in walking robots [21].

Further work in reinforcement learning could also look at ways to increase reward density. This could be done by defining the reward in terms of a sum of desired and undesired effects. In Section 1.2 this was said to not be a good representation of the actual energy efficiency of the flap. But as [21] shows, a reward function of this type can still be used to indirectly encourage improvement of the energy efficiency of a policy. Another option would be to somehow define the energy consumption to be a constant, thus keeping the numerator of the reward ratio constant and allowing the ratio to be split into a sum of ratios with the same numerators; for example by defining the episode to end at once a finite energy buffer is depleted.

The second technique that was discussed uses Fourier series decomposition to define a periodic flapping trajectory. Instead of repurposing the actor critic method from reinforcement learning, optimisation could be done using known black box parameter optimisation techniques such as Bayesian optimisation [8, 19, 14]. This avoid the use of a gradient for optimisation – which could be beneficial when using a ratio as a reward function – and it prevents the need to train neural networks; which is known to not be a very sample efficient task.

A Appendix: Extra runs fourier parameter optimisation

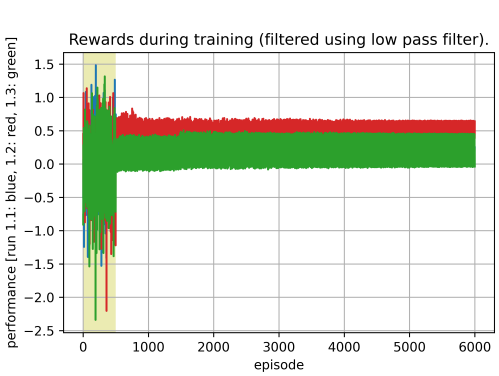
In this appendix, data from extra runs of the Fourier parameter optimisation algorithm from Section 5.4.3 are shown and briefly discussed; Runs are done with a lower noise scalar, with a lower policy learning rate, and with a smaller batch size and higher noise.

A.1 Runs with a lower noise scalar

The runs in Section 5.4.3 use a maximum noise value 0.3 times the maximum parameter value. This value worked well in the earlier experiments where the goal is to minimise per episode energy.

Runs in this section use maximum noise values of 0.1 times the maximum parameter values. As in previous experiments, the scalar of the noise values is decreased linearly during training, until is reaches a value of 0 in the last episode.

Other hyperparameters are kept the same as in Section 5.4.3 (Specifically the runs with low maximum Fourier coefficient values). The small maximum values of the Fourier coefficients ensure that the policy never reaches a OoB state; Therefore, the effects of punishment design can be removed.



(a) Unfiltered rewards during training.

(b) reward data filtered with low-pass (uniform convolution of 30 samples).

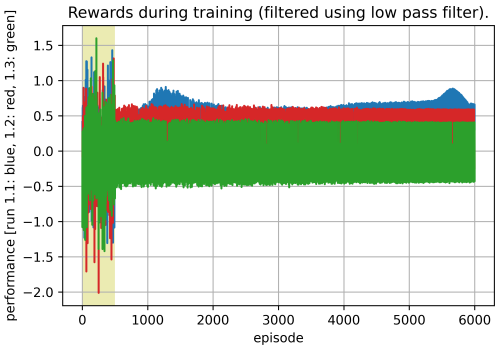
Figure A.1: The rewards during training for all three runs with a lower noise scalar.

None of the 3 runs shows noticeable improvement during training, see Figure A.1. Instead they get stuck in a local optimum soon after finishing the 500 random actions at the start of the episode.

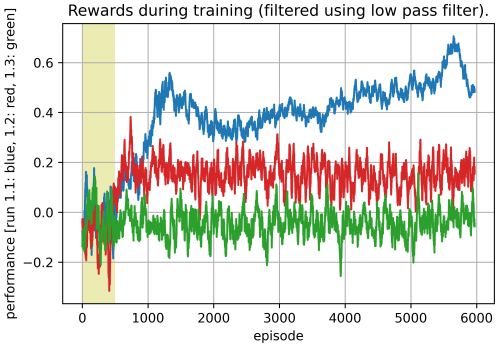
During the periods of stagnation, the gradient of both the Q network and the actor almost completely disappear. Since any new data-points for training Q during this period are based on the current action plus noise, it seems reasonable to assume that these new data-points stop providing novel information about the value function after a certain time of stagnation.

A.2 Runs with a lower policy learning rate

The policy learning rate in Section A.1 is quite high to compensate for lower reward values compared previous experiments. Here, 3 runs are shown with their learning rates decreased from 0.15 to 0.001.



(a) Unfiltered rewards during training.



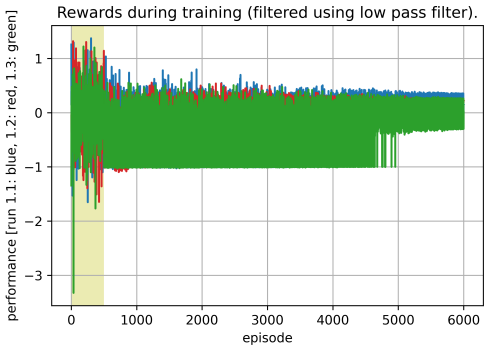
(b) reward data filtered with low-pass (uniform convolution of 30 samples).

Figure A.2: The rewards during training for all three runs with a smaller policy learning rate.

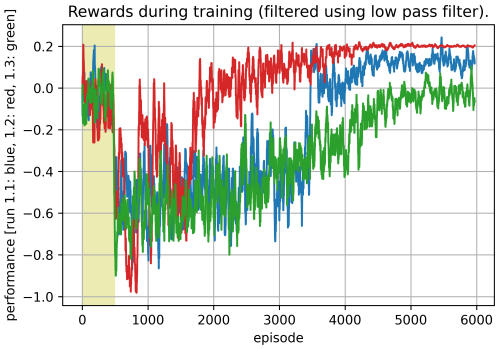
The overall story is similar to previous experiments. The second and third run quickly get stuck and loose their gradients. Run 1 does show improvement, and doesn't lose its critic nor policy gradient, but does not match the random policies in performance. (The best episode in the first run obtains a reward of 1.4 and takes place during the original 500 random policies).

A.3 Runs with higher noise and smaller batch size

Runs in this section use a noise scalar of 0.8 (same as in Section **??**). In addition the batch size is reduced to 64 to match to original DDPG paper.



(a) Rewards during training.



(b) rewards during training, data filtered with low-pass (uniform convolution of 30 samples).

Figure A.3: The rewards during the training episode, simulated plant; Small batch size and high noise.

Again, the story is similar to previous experiments. The runs do show some improvement, but the best performance is far under the performance generated by just trying random policies.

B Appendix: Mathematical model setup

This appendix will outline the mathematics used to model the dynamics of the simple placeholder setup; Section 3.2 and 3.3. The model is fitted to the structure of the "robot equation" [28]:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \tau^T \tag{B.1}$$

Here, the three matrices in this equation represent the inertial, coriolis, and damping effects respectively; the q represent the state vector with q_1 and q_2; and τ represents the motor torques.

B.1 The coordinate system

The system consists of 3 coordinate frames, the world frame ψ_0, the body fixed frame of the first rotational axis ψ_1 and the body fixed frame of the second rotational axis ψ_2 . Each of these axis has a angle q associated with it. The rotation of the first axis is around the x axis of ψ_0, file the rotation of the second axis is around the z axis of ψ_1, see Figure B.1.

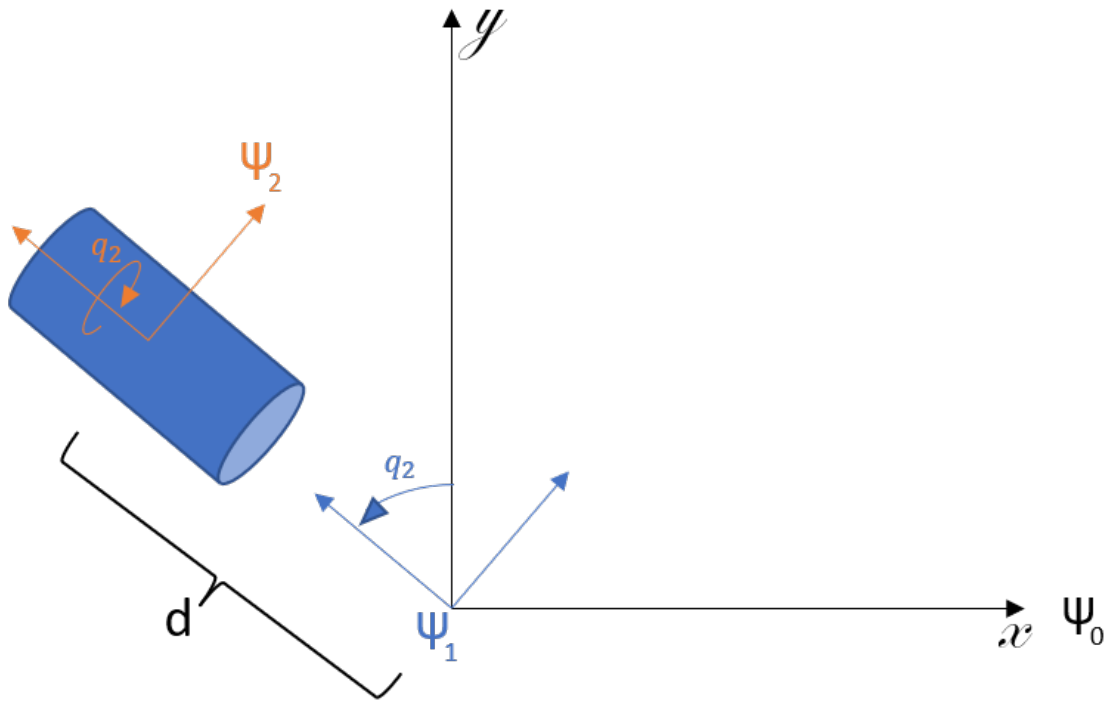


Figure B.1: Geometry of the system in the y-z plane. The x axis of ψ_0 points towards the viewer.

B.2 The I matrix and centre of gravities

All subsequent calculations need information about the inertia matrix of each axis or the position of its centre of gravity.

The inertia matrix can be calculated by integrating over the full mass in in its attached coordinate frame. For the purposes of this model, both intertias are approximated by simple rectangular shapes with homogeneous mass (expressed in meters and kilograms). The total mass and sizes are based on measurements from the real setup. for the wing, this means integrating over a 2 dimensional flat plate; For the servo this means integrating over a off-centre box. This results in the following I matrices:

$$I_{servo,\psi_1} \begin{bmatrix} 3e^{-5} & 0 & 0 & 0 & -0.0004 & 0 \\ 0 & 0.0002 & 0 & 0.0004 & 0 & 0 \\ 0 & 0 & 0.0002 & 0 & 0 & 0 \\ 0 & -0.0004 & 0 & 0.1 & 0 & 0.1 \\ 0.0004 & 0 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.1 \end{bmatrix} \tag{B.2}$$

$$I_{wing,\psi_2} = \begin{bmatrix} 0.003 & 0 & 0 & 0 & -0.02 & 0 \\ 0 & 0.002 & 0 & 0.02 & 0 & 0 \\ 0 & 0 & 0.0005 & 0 & 0 & 0 \\ 0 & 0.02 & 0 & 0.1 & 0 & 0.1 \\ 0.02 & 0 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.1 \end{bmatrix} \tag{B.3}$$

Now the positions of the centres of gravity in both coordinate frames (again according to our simplified masses):

$$P_g^1 = \begin{bmatrix} 0 \\ 0 \\ 0.004 \\ 1 \end{bmatrix} \tag{B.4}$$

$$P_g^2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{B.5}$$

B.3 The M matrix

The M matrix of the system is a sum of M_1 and M_2:

$$M_i(q) = J_i^T(q) Ad_{H_0^i}^T I^i Ad_{H_0^i} J_i(q) \tag{B.6}$$

Here, the J matrix represents the geometric jacobian of the system and the Ad represents th adjoint matrix.

B.3.1 Determining J

The Jacobian of the system can be found by inspection. The first column is simple as it is a simple rotation around the x axis:

$$J_1(q) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{B.7}$$

The second column of the Jacobian is a bit more complicated as the position of the rotational axis is determined by q_1. Note that the rotational axis of q_2 always intersects with the origin of ψ_0; therefore the bottom three values of the second column are all 0.

$$J(q) = \begin{bmatrix} 1 & 0 \\ 0 & -sin(q_1) \\ 0 & cos(q_1) \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{B.8}$$

B.3.2 Determining Ad_0^1

To calculate the Adjoint of ψ_1 matrix, we first need H_0^1, Since the only movement is a rotation around the x axis, it can be written down by inspection:

$$H_0^1(q) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_1 & s_1 & 0 \\ 0 & -s_1 & c_1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B.9}$$

The formulla for the Adjoint is:

$$Ad_{H_i^j} = \begin{bmatrix} R_i^j & o \\ \tilde{p}_i^j R_i^j & R_i^j \end{bmatrix} \tag{B.10}$$

This gives us:

$$Ad_{H_0^1}(q) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & c_1 & s_1 & 0 & 0 & 0 \\ 0 & -s_1 & c_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_1 & s_1 \\ 0 & 0 & 0 & 0 & -s_1 & c_1 \end{bmatrix} \tag{B.11}$$

Multiplying this with J_1 gives us J_1 back, this makes sense as x axis of both the ψ_0 and ψ_1 always overlap:

$$Ad_{H_i^j}(q)J_1(q) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{B.12}$$

B.3.3 Determining Ad_0^2

Ad_0^2 is a bit more complicated. Finding R_0^2 can be done using $R_0^2 = R_1^2 R_0^1$:

$$R_0^2 = R_1^2 R_0^1 = \begin{bmatrix} c_2 & s_2 & 0 \\ -s_2 & c_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_1 & s_1 \\ 0 & -s_1 & c_1 \end{bmatrix} = \begin{bmatrix} c_2 & c_1 s_2 & s_1 s_2 \\ -s_2 & c_1 c_2 & s_1 c_2 \\ 0 & -s_1 & c_1 \end{bmatrix} \tag{B.13}$$

P_0^2 can be found by inspection. d represents the distance between the two coordinate frames (see Figure B.1). P_0^2 can be described as follows; the coordinates of the base of ψ_0 in ψ_2. Since ψ_2 is always oriented towards the origin of ψ_0 with its z axis (fig. B.1), P_0^2 becomes a constant:

$$p_0^2 = \begin{bmatrix} 0 \\ 0 \\ -d \end{bmatrix} \tag{B.14}$$

Using this information the Adjoint matrix can be constructed and multiplied with the Jacobian. This is written out as it does not provide any insight.

B.4 The C matrix

The C matrix can be calculated straight from the M matrix using:

$$C_{i,j}(q,\dot{q}) = \Gamma_{i,j,k}\dot{q}^k \tag{B.15}$$

$$\Gamma_{i,j,k} := \frac{1}{2}\left(\frac{\partial M_{ij}}{\partial q_k} + \frac{\partial M_{ik}}{\partial q_j} - \frac{\partial M_{kj}}{\partial q_i}\right) \tag{B.16}$$

The K represents the index of the coordinate vector $q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$. Again, the full matrix is not written out here as it is not very insightfull.

B.5 The G vector

The G matrix is the derivative of the potential energy over the generalised coordinates:

$$G = \frac{\partial V}{\partial q} \tag{B.17}$$

The potential energy can be described as the sum of the potential energy of each rigid body:

$$V(q) = V_1(q) + V_2(q) \tag{B.18}$$

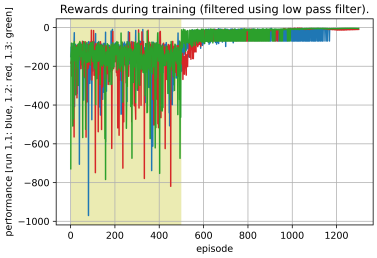
Here, the potential energy for each frame is calculated from the previously calculated values as:

$$V_i(q) = m_i g (0\,0\,1\,0) H_i^0(q) P_g^i \tag{B.19}$$

Converting H_0^i to H_i^0 is done using:

$$\begin{aligned} H_i^0 = (H_0^i)^{-1} &= \begin{bmatrix} (R_0^i)^T & -(R_0^i)^T o_0^i \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} & 1 \end{bmatrix}, \\ \text{for: } \quad H_0^i &= \begin{bmatrix} R_o^i & o_0^i \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} & 1 \end{bmatrix} \end{aligned} \tag{B.20}$$

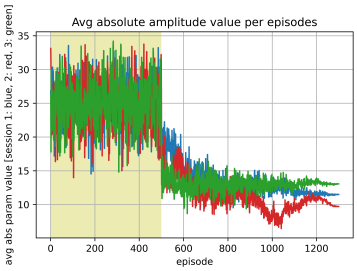
C Appendix: raw data from experiment energy minimisation



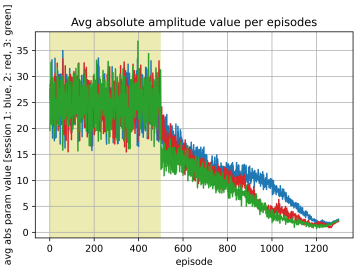
(a) Digital setup.

(b) Real setup.

Figure C.1: Raw reward data from Section 5.4.2.



(a) Digital setup.



(b) Real setup.

Figure C.2: Raw reward data from Section 5.4.2.

Bibliography

[1] (2016), *Mechaduino - Powerful open-source industrial servo motor. by Tropical Labs — Kickstarter.*, https://www.kickstarter.com/projects/tropicallabs/mechaduino-powerful-open-source-industrial-servo-m [Accessed: August 2022].

[2] (publication date unknown), *Adam — PyTorch 1.12 documentation*, https://pytorch.org/docs/stable/generated/torch.optim.Adam.html [Accessed: August 2022].

[3] (publication date unknown), *DFRobot DSS-M15S servo*, https://www.dfrobot.com/product-1709.html [Accessed: August 2022].

[4] (publication date unknown), *scipy.integrate.ode — SciPy v1.8.1 Manual*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html [Accessed: August 2022].

[5] (publication date unknown), *Servo - Arduino Reference*, https://www.arduino.cc/reference/en/libraries/servo/ [Accessed: August 2022].

[6] (publication date unknown), *The portwings project*, http://http://www.portwings.eu/.

[7] Angelini, F., C. Della Santina, M. Garabini, M. Bianchi and A. Bicchi (2020), Control architecture for human-like motion with applications to articulated soft robots, *Frontiers in Robotics and AI*, p. 117.

[8] Bergstra, J., R. Bardenet, Y. Bengio and B. Kégl (2011), Algorithms for hyper-parameter optimization, *Advances in neural information processing systems*, **vol. 24**.

[9] Betts, J. (2001), Practical methods for optimal control using nonlinear programming, ser, *Advances in Design and Control. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM)*, **vol. 3**.

[10] Bøhn, E., E. M. Coates, S. Moe and T. A. Johansen (2019), Deep reinforcement learning attitude control of fixed-wing uavs using proximal policy optimization, in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, pp. 523–533.

[11] Della Santina, C., M. Bianchi, G. Grioli, F. Angelini, M. Catalano, M. Garabini and A. Bicchi (2017), Controlling soft robots: balancing feedback and feedforward elements, **vol. 24**, no.3, pp. 75–83.

[12] Diehl, M., H. G. Bock, H. Diedam and P.-B. Wieber (2006), Fast direct multiple shooting algorithms for optimal robot control, in *Fast motions in biomechanics and robotics*, Springer, pp. 65–93.

[13] Emken, J. L., R. Benitez, A. Sideris, J. E. Bobrow and D. J. Reinkensmeyer (2007), Motor adaptation as a greedy optimization of error and effort, **vol. 97**, no.6, pp. 3997–4006.

[14] Frazier, P. I. (2018), A tutorial on Bayesian optimization, *arXiv preprint arXiv:1807.02811*.

[15] Haarnoja, T., A. Zhou, P. Abbeel and S. Levine (2018), Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, in *International conference on machine learning*, PMLR, pp. 1861–1870.

[16] Hwangbo, J., I. Sa, R. Siegwart and M. Hutter (2017), Control of a quadrotor with reinforcement learning, **vol. 2**, no.4, pp. 2096–2103.

[17] Iscen, A., K. Caluwaerts, J. Tan, T. Zhang, E. Coumans, V. Sindhwani and V. Vanhoucke (2018), Policies modulating trajectory generators, in *Conference on Robot Learning*, PMLR, pp. 916–926.

[18] Kelly, M. P. (2017), Transcription methods for trajectory optimization: a beginners tutorial, *arXiv preprint arXiv:1707.00284*.

[19] Knysh, P. and Y. Korkolis (2016), Blackbox: A procedure for parallel optimization of expensive black-box functions, *arXiv preprint arXiv:1605.00998*.

[20] Koch, W., R. Mancuso, R. West and A. Bestavros (2019), Reinforcement learning for UAV attitude control, **vol. 3**, no.2, pp. 1–21.

[21] Lee, J., J. Hwangbo, L. Wellhausen, V. Koltun and M. Hutter (2020), Learning quadrupedal locomotion over challenging terrain, **vol. 5**, no.47, p. eabc5986.

[22] Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra (2015), Continuous control with deep reinforcement learning, *arXiv preprint arXiv:1509.02971*.

[23] Matheron, G., N. Perrin and O. Sigaud (2019), The problem with DDPG: understanding failures in deterministic environments with sparse rewards, *arXiv preprint arXiv:1911.11679*.

[24] Meinsma, G. and A. v. d. Schaft (2020), *Calculus of Optimal Control (lecture notes)*, Faculty of electrical engineering mathematics, and computer science, University of Twente.

[25] Schulman, J., S. Levine, P. Abbeel, M. Jordan and P. Moritz (2015), Trust region policy optimization, in *International conference on machine learning*, PMLR, pp. 1889–1897.

[26] Schulman, J., F. Wolski, P. Dhariwal, A. Radford and O. Klimov (2017), Proximal policy optimization algorithms, *arXiv preprint arXiv:1707.06347*.

[27] Soliman, S. S. and M. D. Srinath (1998), *Continuous and Discrete Signals and Systems*, Pearson, ISBN 9780135184738.

[28] Stramigioli, S. (2020), Lecture slides on Modern Robotics.

[29] Sutton, R. S. and A. G. Barto (2018), *Reinforcement learning: An introduction*, MIT press.

[30] Tang, C. and Y.-C. Lai (2020), Deep reinforcement learning automatic landing control of fixed-wing aircraft using deep deterministic policy gradient, in *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, pp. 1–9.