POLITECNICO DI TORINO

Master's Degree in Aerospace Engineering

von KARMAN INSTITUTE
FOR FLUID DYNAMICS

Master's Degree Thesis

On the Model Based and Model Free Control of a Flapping-Wing Micro Air Vehicle

Supervisors

Prof. Sandra PIERACCINI

Prof. Miguel Alfonso MENDEZ

Co-Supervisors

Lorenzo SCHENA

Romain POLETTI

Candidate

Alberto Maria CRAVERO

July 2025

Abstract

Flapping-Wing Micro Air Vehicles (FWMAVs) are micro-drones bio-inspired by birds and insects. They possess a unique way to generate lift and thrust via flapping wing motions, which provide exceptional maneuverability. This peculiar flight mechanism makes them suitable for tasks such as search and rescue, surveillance and environmental monitoring. However, it also introduces significant challenges in modeling their dynamics and aerodynamics due to strong nonlinearities.

Given these complexities, developing effective control algorithms for FW-MAVs is a challenging problem. Traditional control theory aims to compute control sequences that guide a system along a desired trajectory. While possible, applying these methods to highly nonlinear systems can be difficult. Alternatively, Reinforcement Learning (RL), subfield of machine learning born in the 90s, focuses on learning a control policy with limited or no prior knowledge of the system's dynamics. Recent advances in RL have led to Deep Reinforcement Learning (DRL), which employs neural networks and other function approximators to learn complex control strategies. However, training these models can be computationally expensive and difficult to tune: while more complex architectures and algorithms are being created, we wonder if simpler and more direct architectures can perform comparably or even better in certain scenarios.

This work aims to evaluates the performance of Radial Basis Functions (RBFs) as function approximators to model the control policy in two test cases. In the first, simplified test case, the RBF-based controller will be compared with a Linear Quadratic Regulator (LQR), designed using optimal control theory. This scenario considers a one-dimensional, averaged model of the drone's motion, where the objective is to hover at a fixed target position. The second test case is more realistic, relying on the full nonlinear dynamics of the system and explores the learning challenges faced by the RBF-based controller still within the context of one-dimensional hovering control.

Acknowledgements

This journey was not the easiest, but probably the one I needed. Hoping that I have finally found my path, I want to express my most sincere gratitude to those who have supported me along the way.

First, my family: your presence and belief in me have been my foundation through all these years. To my beloved Claudia, my constant source of inspiration and strength in the toughest moments.

I extend my heartfelt thanks to my advisors, Miguel Mendez and Sandra Pieraccini, for their guidance and for giving me the opportunity to pursue this thesis.

A special mention goes to my co-supervisors, Lorenzo Schena and Romain Poletti, whose support and insights have been invaluable throughout this work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Control System

In engineering and science, control problems involve designing strategies or algorithms to govern the behavior of a dynamical system, ensuring that it operates in a desired manner [1],[2]. The system, often governed by complex physical or mathematical dynamics, interacts and responds to inputs or actions from the controller. The challenge lies in determining the optimal sequence of actions that achieve a specific objective while respecting constraints and minimizing undesired outcomes.

At the core of a control problem is a feedback loop, where the system's state (e.g., position, velocity, temperature, ...) is monitored and used to adjust control actions. The goal is often to drive the system toward a desired target state, maintain stability, or optimize performance. For instance, in robotics, control problems might involve making a robotic arm precisely reach a target position, while in aerospace it could involve stabilizing a drone in turbulent conditions while following a trajectory.

Formally, the control problem can be framed as an optimization task where the objective is to minimize a cost or maximize a reward. This reward can be defined in terms of how close the system's state is to a target or how efficiently the system achieves its goals. For example, the cost might represent the squared error between the system's current state and a desired target state, while constraints could enforce limits on control inputs or system behavior.

Controllers can be designed using various approaches, including classical

techniques like Frequency Response Analysis or modern strategies that leverage machine learning and reinforcement learning. The latter consists in learning control policies that can map system's states to actions directly, often through interaction with a simulated or real environment. These methods, composing the field of *Control Learning*, are particularly useful for complex, high-dimensional systems where deriving analytical solutions is impractical and model-based methods are inaccurate due to model limitations.
Finally, solving a control problem requires a balance between robustness, precision and computational efficiency, in order to ensure that the system performs reliably across a range of conditions.

1.2 FWMAVs

Flappin-Wing Micro Air Vehicles (FWMAV [3], [4]) are flying, bio-inspired robots that reproduce the flight mechanism of birds and insects. Their unique way of fly is characterized by the oscillatory motion of their wings to generate both lift and thrust, enabling them to hover, glide, and perform agile maneuvers in confined spaces. These micro-drones typically have wingspans ranging from 1cm to 35cm, masses up to 80g and flight ranges of up to 1km; thanks to their highly maneuverability, they're suitable for various task such as search and rescue missions, environmental monitoring, surveillance, and planetary exploration. FWMAVs are built with lightweight materials and flexible wings, along with advanced control systems that help them adjust in real time for optimal flight. Challenges include improving lift-to-weight ratios, creating efficient power sources and strong controllers that can handle the unpredictable nature of flapping flight. In fact, this particular way of flying introduces different non-linearities (such as unsteady aerodynamics and strong coupling between wing motion and body response) that lead to significant challenges in modeling their dynamics and thus in developing efficient and suitable control strategies. Modern control theory, which aims to find a correct control sequence (with respect to some measure) to drive the system along a certain trajectory, faces big challenges when dealing with this kind of systems: as the non-linearities increase, classical laws become less effective and modeling is more troublesome.

Figure 1.1: Flapping-Wing Micro Air Vehicle, from Keennon et al. [4]

As technology improves, FWMAVs are likely to become important in robotics, providing new solutions for tasks that require high flexibility and adaptability in limited spaces.
In the context of control learning, it becomes interesting to see whether machine learning methods can overcome non-linearity and uncertainties and produce controllers capable of meeting the necessary requirements.

1.3 Goals

This thesis has two main goals: first, it aims to compare model based control methodologies (Linear Quadratic Regulator optimal control) and model-free approaches in terms of both optimality and robustness. To do so, a first oversimplified test case based on averaged dynamics will be used. This simplification enables the development of the model-based framework while avoiding the troublesome non-linearities. On the other hand, the model free controller will be directly optimized using a state-of-art optimizer in conjunction with a particular neural architecture, known as Radial Basis Function, to model the policy. These two approaches will be compared in terms of performance and robustness to noise.
The second objective of this thesis is to assess the capabilities of the model-free controller in a more realistic setting, by testing it against a full flapping dynamics model that takes into account the aerodynamic effects of flapping wing flight. To do so, the Radial Basis Function policy will be

trained using model free reinforcement learning algorithms: in particular, the ease of training this policy approximator will be investigated using simplified versions of policy gradient methods.

1.4 Thesis outline

The thesis is organised as follows:

- Chapter 2 addresses the methodologies. In this second section, the control problem is first formally introduced and a taxonomy of control strategies presented. Then, the (modern) optimal framework is discussed. Ultimately, we talk about reinforcement learning and function approximators.
- Chapter 3 describes the two test cases, the averaged and the full-dynamics environments, and derives their equations of motion.
- Chapter 4 analyzes the model-based approach, which involves the development of an LQR controller.
- Chapter 5 analyzes the model-free approach, focusing on the implementations of RBF controllers.
- Chapter 6 presents the results.

Chapter 2

Methodologies

In this section, the methodologies for this work are introduced. First, the general statement of a control problem is given, together with some notions about different types of control. Then, the two techniques used (optimal control and reinforcement learning) are described.

2.1 Control problem statement

Starting with an ordinary differential equation (ODE)

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t)) \qquad (t > 0) \\ \mathbf{x}(0) = \mathbf{x}_0 \end{cases} \tag{2.1}$$

We are given the initial point $\mathbf{x}_0 \in \mathbb{R}^n$ and the function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. The unknown is the curve $\mathbf{x}(t) : [0, \infty) \to \mathbb{R}^n$, which we interpret as the dynamical evolution of the state of some “system”. Supposing that $\mathbf{f}$ depends also on some "control" variable $\mathbf{u}$ belonging to a set $\mathrm{U} \subset \mathbb{R}^m$ so that $\mathbf{f} : \mathbb{R}^n \times \mathrm{U} \to \mathbb{R}^n$, we get a dynamical system that is subjected on $\mathbf{u}$. By considering $\mathbf{u}(t) : [0, +\infty) \to \mathrm{U}$ as function of time, we get

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \qquad (t > 0) \\ \mathbf{x}(0) = \mathbf{x}_0 \end{cases} \tag{2.2}$$

where the *response* or *trajectory* $\mathbf{x}(\cdot)$ is subjected on the *control* $\mathbf{u}(\cdot)$. The goal, is to identify $\mathbf{u}$ so that the system follows the target trajectory, defined as $\mathbf{x}_* : [0, \infty) \to \mathbb{R}^n$. There are two different approaches to do so:

- **Open Loop**: A type of system control that works without any feedback. It follows a predefined sequence of inputs u, without taking in consideration the actual output or system state. By not using a feedback, it cannot correct its errors or compensate for any disturbances.

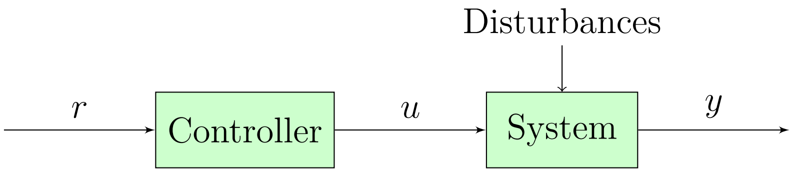


Figure 2.1: Open loop diagram. r is the reference signal, u is the control, y is the system's output.

- **Closed Loop** or feedback control: is a type of control that monitors and adjust continuously its output based on a feedback $e = r - y_m$ from the system that is controlling, reacting to disturbances and changes in the environment.

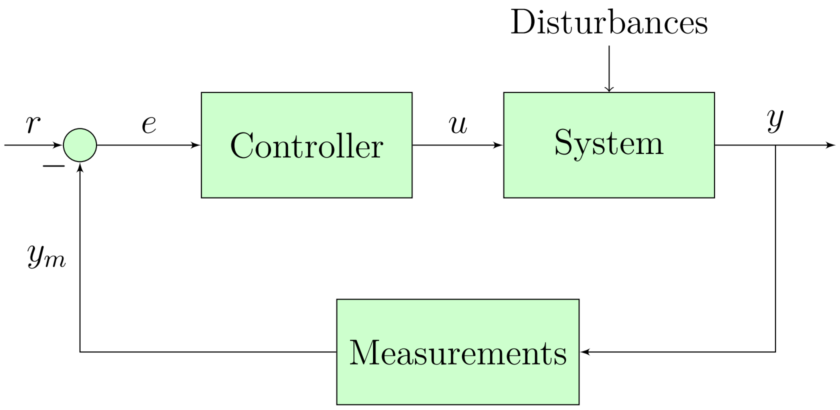


Figure 2.2: Closed loop diagram. e is the control error and y_m is the measured output.

While easier to design, open loop control can't guarantee any form of stability. Therefore we will focus on closed loop controls, which can also be divided in:

- **Classic control**: Classic or conventional control, focuses itself on linear SISO (single-input, single output) system. It works primary in the frequency domain, with methods such as Laplace transforms and transfer function. Common techniques are Root Locus [5] for stability, Bode and Nyquist Plots to analyze performance and stability and Proportional-Integral-Derivative[1] (PID, [6]) for error correction.

- **Modern control**: It extends to MIMO (multi-input, multi-output) systems and it works in the time domain. Rather than transfer function, it leverages on state-space representation and it's useful for non-linear system and optimal control application. While classical controllers are still widely used in practice due to their simplicity and effectiveness, modern control techniques offer a more general framework for analyzing and designing controllers for complex, high-dimensional, or uncertain systems.

In this work we will focus on the latter. Beside optimal control, discussed in Section 2.2, modern techniques are:

- **Adaptive control**: the control method used by a controller which must adapt to a system with varying or initially uncertain parameters, such as a flying aircraft which mass slowly decreases as result of fuel consumption. A law that can adapt itself to such conditions is needed, and differently from robust control it doesn't need a priori information about bounds or time-varying parameters: while robust control guarantees that, under certain circumstances, the law doesn't have to be changed, adaptive control designs laws that change themselves.
- **Robust control**: a control method that explicitly deals with uncertainty, rather than adapting itself. This approach is designed to function properly under the assumption that uncertain parameters and/or disturbances are found within some set. Robust control aims to achieve *robust* performances and stability in the presence of bounded modelling errors. The most known example is H_∞ loop-shaping [7]

[1] PID controllers are widely used in classical control due to their simplicity and effectiveness in many industrial applications: they compute the control input as a weighted combination of the system's state error, its integral and its derivative. These gains can be tuned with various techniques, even machine learning based, in order to achieve stability, improve transient response and reduce steady-state error.

which minimizes the system's sensitivity over its frequency spectrum and guarantees that the system will not deviate from expected trajectories under disturbances. Sliding mode control (SMC) [8] is another important technique, which ensures great stability properties while possessing particular simplicity in its design.

- **Model Free**: with the advent of machine learning, this new type of control emerged. Model free methods do not require an explicit mathematical model of the controlled system: it tries to learn control policy directly from data instead with a trial-and-error approach. This work will explore this type of methods in Section 2.3.

2.2 Optimal Control

Optimal control is a framework in which the control inputs are designed to optimize a given objective, while satisfying the system's dynamics and sets of constraints. The problem is typically formulated as maximizing (or minimizing) a cost functional J which measure performance, subject to the system's differential equation (2.2) and constraints on trajectories and control. Different tools are available to solve optimal control problems, such as calculus of variations, dynamic programming and numerical optimization. It's widely applied in fields such as engineering, robotics and economics, where system must function efficiently, and applications range from energy-efficient control of industrial processes to autonomous vehicles' trajectory planning.

The development of optimal control theory has seen to two major approaches:

- the **Pontryagin Maximum Principle** [9], which provides necessary conditions for optimality using the Hamiltonian and adjoint variables
- the **Hamilton-Jacobi-Bellman Equation**, which offers a recursive, dynamic programming-based approach to determine optimal control strategies.

Both arises from the calculus of variations and give complementary perspectives on how to solve optimal control problems. While PMP is usually employed in open-loop solutions, HJB equation naturally suits closed-loop strategies, making it very useful in modern control and reinforcement learning and will be therefore explored in this work.

2.2.1 Dynamic Programming and the Principle of Optimality

Dynamic programming [10] is a powerful technique, primarily used for optimization problems, used to solve complex tasks by breaking them into simpler and overlapping subtasks. Its foundation in the contest of optimal control is given by Bellman's Principle of Optimality, which states:

> An optimal policy has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with respect to the state resulting from the first decision.

In mathematical terms, this principle leads to a recursive relationship for an optimal cost-to-go function. To develop this equation, commonly known as the Bellman equation, let's recall the dynamical system from (2.2):

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(\mathbf{x}, t)), \quad \mathbf{x}(t_0) = x_0, \tag{2.3}$$

where $\mathbf{x}(t)$ is the system state, $\mathbf{u}(t)$ is the control input, and $\mathbf{f}$ describes the system dynamics.
The performance of a control input $\mathbf{u}(\cdot)$ over the time interval $[t_0, t_1]$ is evaluated using the cost functional:

$$J(\mathbf{u}) = \int_{t_0}^{t_1} \tilde{J}(t, \mathbf{x}(t), \mathbf{u}(t))dt + K(\mathbf{x}(t_1)), \tag{2.4}$$

where $\tilde{J}$ is the running cost and K is the terminal cost. The goal is to find a control policy $\mathbf{u}^*$ that minimizes J.

2.2.2 The Hamilton-Jacobi-Bellman Equation

To address this minimization, we define the *value function*:

$$V(t, \mathbf{x}) = \inf_{\mathbf{u}_{[t,t_1]}} J(t, \mathbf{x}, \mathbf{u}), \tag{2.5}$$

which represents the optimal cost-to-go from state $\mathbf{x}$ at time t. The boundary condition for V comes from the terminal cost:

$$V(t_1, \mathbf{x}) = K(\mathbf{x}) \tag{2.6}$$

By applying the principle of optimality over a small interval $[t, t + \Delta t]$, the dynamic programming equation is derived:

$$V(t, \mathbf{x}) = \inf_{\mathbf{u}_{[t,t+\Delta t]}} \int_t^{t+\Delta t} \tilde{J}(s, \mathbf{x}(s), \mathbf{u}(s))ds + V(t + \Delta t, \mathbf{x}(t + \Delta t)) \tag{2.7}$$

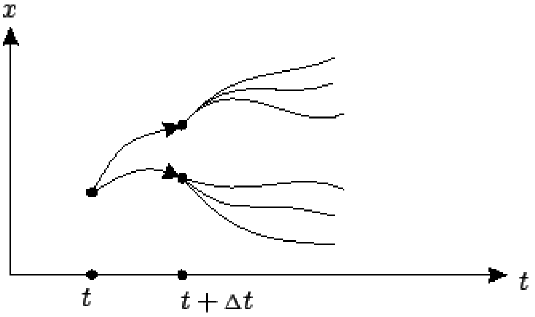


Figure 2.3: Principle of optimality in continuous time.

Expanding $V(t + \Delta t, \mathbf{x}(t + \Delta t))$ using a first-order Taylor expansion and dividing by Δt, we obtain:

$$\frac{\partial V}{\partial t} + \min_u \left[\tilde{J}(t, \mathbf{x}, \mathbf{u}) + \nabla V \cdot \mathbf{f}(\mathbf{x}, \mathbf{u})\right] = 0. \tag{2.8}$$

This partial differential equation is the Hamilton-Jacobi-Bellman (HJB) equation, which is a powerful tool for solving optimal control problems as it characterizes the value function $V(t, \mathbf{x})$ explicitly; both (2.7) and (2.8) provide necessary and sufficient conditions for optimality.

2.3 Reinforcement Learning

Three basic machine learning paradigms exists:

- **Supervised Learning**: the learning is based on expert-labelled data, which provide additional information to the algorithm. For instance, this type of ML could be used to calculate a regression with a set of data marked so as to correspond to a user-defined mathematical criterion.
- **Unsupervised Learning**: the data used to learn is not labelled before-hand. Thus, some criteria have to be defined in order to extract the features directly from data. This approach is used for example for clustering and dimensionality reduction.
- **Reinforcement Learning (RL)**: aims to train an intelligent agent to take actions in a dynamic environment in order to maximise (or minimise) a certain reward (or cost); here the data is not available but has to be generated by the agent interacting with the environment

The goal of a RL agent is to maximise expected reward over it's lifetime. What the agent experiences over it's lifetime, i.e. rewards, states and actions defines it's trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, ..., r_{T-1}, s_{T-1}, a_{T-1}, r_T, s_T)$. The trajectories that an agent might experience depend on what actions a_t it takes from any given state s_t following the policy π; thus the environment evolves to a new state s_{t+1} producing a reward r_{t+1}.

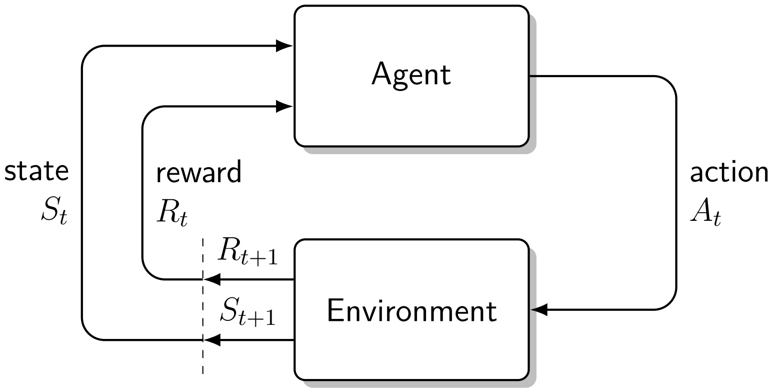


Figure 2.4: RL scheme.

The policy π is a function that maps states into actions and models the agent's action selection process; it can be

- **deterministic**: $\pi : \mathcal{S} \rightarrow \mathcal{A}$, mapping from state space into action space, the output is a parameter
- **stochastic**: $\pi : \mathcal{S}$ x $\mathcal{A} \rightarrow$ [0,1], the output is a parameter of a probability density function.

The reward r_t is obtained after each action taken by the agent. The goal of the agent is to minimise the total reward obtained during a trajectory

$$R_T = \sum_{t=0}^{T} r_t. \tag{2.9}$$

or its discounted version

$$R_T = \sum_{t=0}^{T} \gamma^t r_t. \tag{2.10}$$

The term γ^t ensures that immediate rewards are preferred to future ones. Furthermore, if the episode has infinite length (i.e. $T = \infty$), this term ensures that the total reward R_T is still finite.

2.3.1 Model Based vs Model Free Reinforcement Learning

There are two main branches in Reinforcement Learning, which differ if the agent has access or learns a model of the environment (e.g. something that can predict rewards and state transitions): if this model exists, the algorithm is called model-based.
The main advantage of having a model is that it allows the agent to plan in advance, knowing what would happen for a range of possible choices and explicitly deciding between its options. Agents can then turn the results of this planning into learning a policy. One particularly famous example is AlphaGo [11], which was the first software to win against a human without handicaps. When a model of the system is somehow accessible, it can result in a substantial improvement in the efficiency of sampling compared to model free methods. However, a ground-truth

model of the environment is usually not available. If an agent wants to use a model, it usually has to learn it purely from experience, which creates several challenges as well. The biggest problem is that any bias in the model can and will be exploited by the agent, resulting in good performance with respect to the learned model and at least sub-optimal (if not terrible) performance in the real environment, a phenomena called *over-fitting*. Thus, model learning is hard and usually fail to pay off.
Algorithms that don't use a model are called model-free: while losing the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune. This kind of methods will be explored in the thesis.
There are two main approaches to train agents within model-free RL: Policy based and Value based algorithms.

2.3.2 Value based algorithm

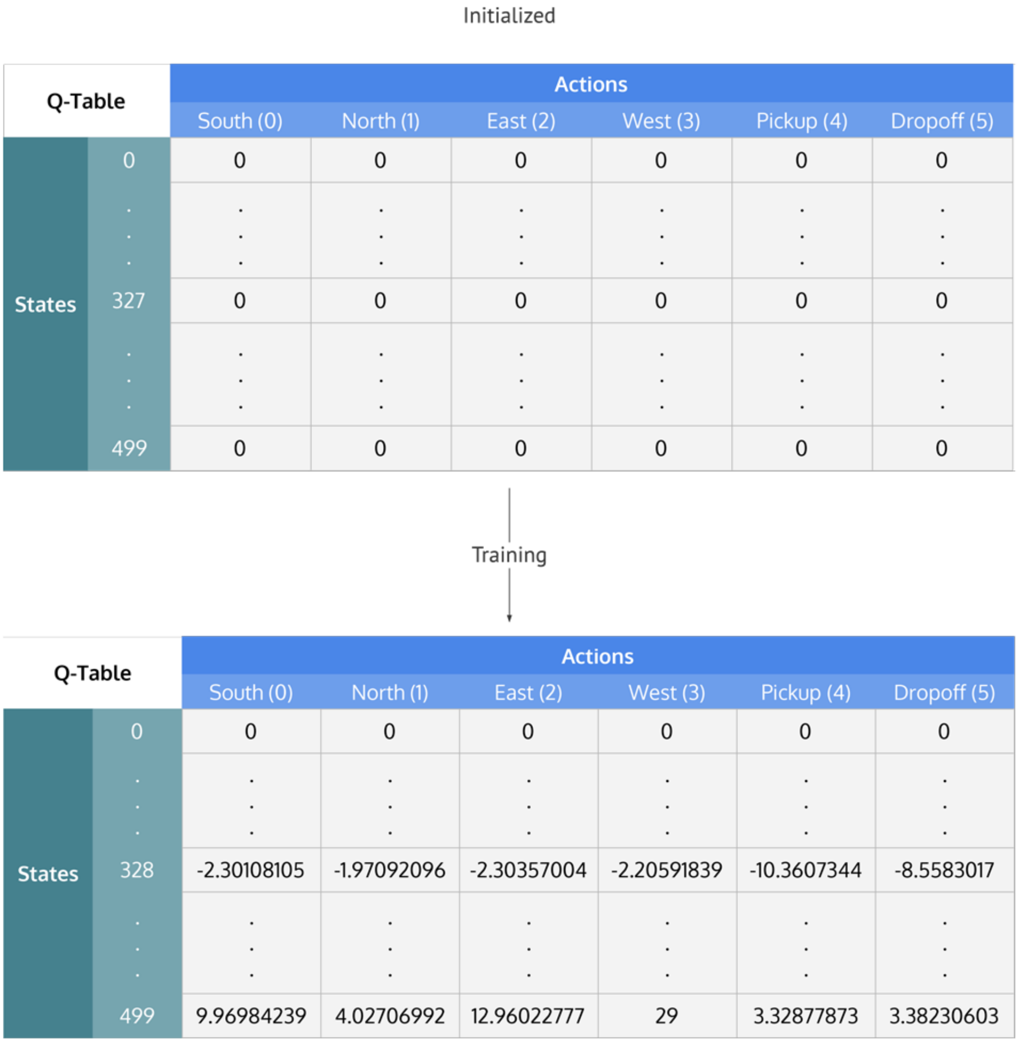
The first tier of RL algorithm are the value-based one. In order to discover the best policy, the idea is to learn some sort of function that tells how good is to take a certain action in a certain state. This function is called action-value function or *q-function* $Q(s, a)$, and corresponds for the pair (s, a) to the expected return taking action a starting from state s, and from then on following policy π (thus considering the expectation of future rewards):

$$Q(s,a) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t | s_0 = s, a_0 = a\right] \tag{2.11}$$

After estimating Q, the optimal policy is then obtained just by selecting each state the action with the highest $Q(s, a)$ in a setting known as ϵ-greedy. One of the most known algorithm in this framework is *Q-learning* which uses a table such as Figure 2.5 to store $Q(s, a)$ and updates it every step by using iteratively the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right], \tag{2.12}$$

where α is the learning rate, r is the reward obtained, γ is the discount factor (2.10) and $\max_{a'} Q(s', a')$ represents the highest expected value for the state s'. This way, it's possible to learn an accurate estimation of the actions' values without explicitly knowing the environment, while the optimality is ensured thanks to the Bellman equation.


Initialized
Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0	0	0

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 2.5: Q-Table update for discrete action and state space.

An advantage is that these type of algorithms are off-policy, which means that the policy improved during training is different from the one used for action selection. However, a *finite* tabular representation works only for *discrete* state and actions; when the state and/or action spaces become continuous it is necessary to discretize, and when they're too large using a table becomes impracticable. Therefore a different representation for Q is needed, and here function approximators such as Neural Networks (NN) come to play: Deep Q-Network (DQN, [12]) leverages on a deep neural network in order to model $Q(s, a)$. DQN training is done by minimizing the square error between the network output for a pair (s, a) and the target one computed using Bellman equation. In order to improve stability and efficiency of training, there are different techniques such as *replay buffer*, which stores past transitions for random sampling, and *target network*, which is updated more slowly to provide stable targets. These tricks have made the DQN a powerful tool for solving complex problems such as those of video games, with results comparable or superior to those of human beings in some cases.

2.3.3 Policy based algorithm

Policy based methods use stochastic policies, explicitly representing them as probability distributions $\pi_\theta(a|s)$ without modeling the action-value function. This algorithms optimize the parameters θ either directly by gradient ascent using the performance objective $J(\pi_\theta)$, or indirectly, by maximizing surrogates that usually give conservative estimates of how much the performance objective $J(\pi_\theta)$ will change after the update. Usually the optimization is carried out on-policy, which means that the data used for each update are collected while acting according to the latest version of the policy. While being computationally effective, this approach is also dangerous: if the agent learns bad policies, it can be quite difficult to recovery[2] To face this problem a new family of algorithms, called trust region methods, have been developed. Policy based algorithms are further discussed in (5.1)

2.3.4 Actor critic framework

In the Actor-critic framework, policy optimization also involves learning an approximator $V^\psi(s)$ for the on-policy value function $V^\pi(s)$. The idea, represented in Figure 2.6, is to split the learning in two parts:

[2] This issue is called sub-optimal policy learning.

- the **Actor** has to model the agent's behavior. It computes the action taking in input the state, according to the specific algorithm that is being used, thus learns either directly the policy or the q-function $Q(s,a)$.

- the **Critic** has to learn $V_x(s)$ and computes the value function throughout the training. Its input is the action chosen and the output is the value, which corresponds to its maximum future reward.

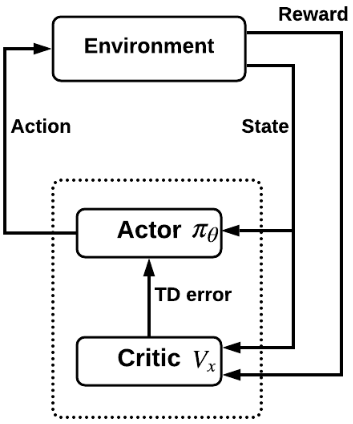


Figure 2.6: Actor-Critic scheme.

Different function approximator can be used to model both the Actor and the Critic (see Section 2.3.5); the training of the nets is performed separately and the weights (in this work θ for the Actor and ψ for the Critic) are updated at each time step. Among the most popular Actor-Critic methods there are Deep Deterministic Policy Gradient DDPG [13], Soft Actor-Critic SAC [14], Advantage Actor-Critic A2C [15] and Proximal Policy Optimization PPO [16].

2.3.5 Neural network and approximators

Depending on the algorithm, different tools are to be used in order to model the Q-function, the value function or the policy. For instance in Q-learning, when dealing with simple and discrete state-action space, the *q-table* is sufficient but when the state space is continuous, it is necessary

to employ a more sophisticate way to model it.
For this purpose different function approximators can be used, whereas artificial neural networks are the most common choice.

- **ANN**: Artificial neural networks are the fundamental building blocks of deep learning. They are multi-layered networks of neurons, mathematical functions that take some inputs x, weight them and then sum them up to produce an output y. Mathematically, a neuron reads:

$$y = f\left(\sum_i w_i x_i + b\right) \tag{2.13}$$

  The input received through the input layer of each neuron, summed and weighted, are processed throughout the activation function f (which usually is a sigmoid or an inverse tangent) to produce the output. Then, different neurons are put together forming a hidden layer, that takes in input the output of the previous one (or the input layer itself) and produces outputs for the next one. Neural networks are universal "black-box" approximators, since any function can be modeled and approximated by a neural network with the proper structure. Thus different structures exist, such as MLP, LTSM, Neural ODE, and serve different purposes.

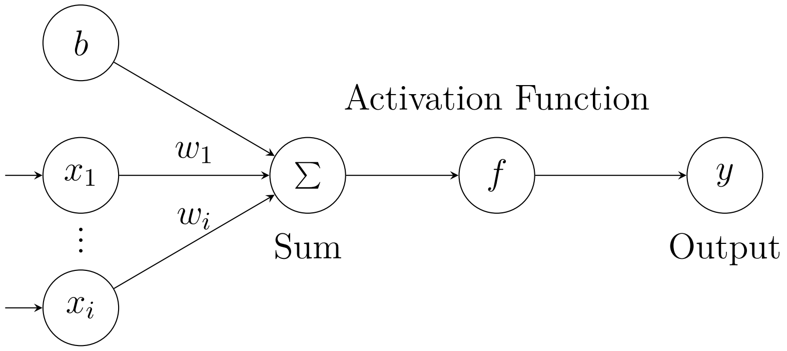


Figure 2.7: Generic neuron $\phi_j^{(n)}$ representation.

- **MLP**: Multilayer Perceptron [17] is a type of feedforward artificial neural network that possess at least three layers: one input layer, one or more hidden layers, and one output layer. Each layer is fully connected to the next one, which means that each neuron in the previous layer is connected to every neuron in the next layer. MLPs use nonlinear activation functions (like ReLU, and sigmoids), and are able to learn complex mappings between inputs and outputs. They are widely applied in tasks like regression, image classification, and time series forecasting.

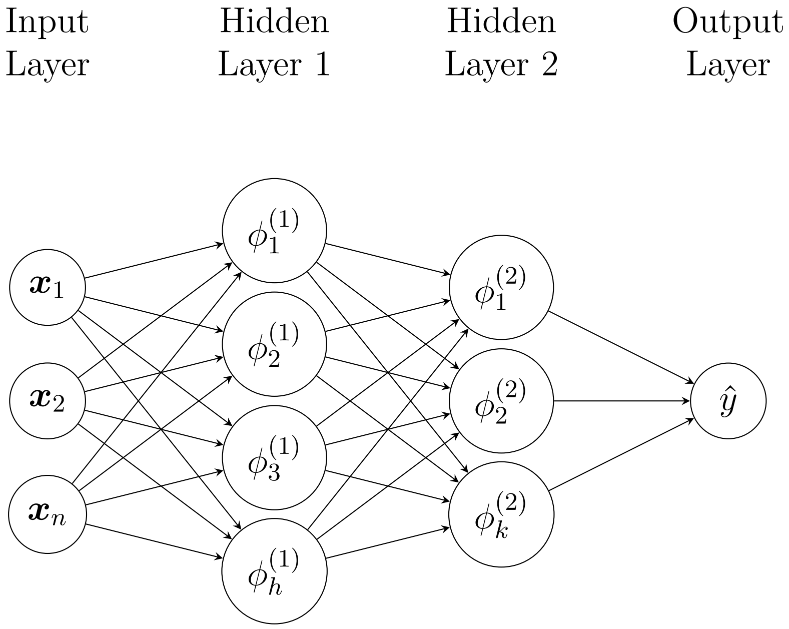


Figure 2.8: MLP schematics.

- **LTSM**: Long Short-Term Memory [18] is a recurrent neural network (RNN) architecture specifically suited to handle sequential data and overcome the vanishing gradient problem[3] that is often encountered when using traditional RNNs. LSTMs introduce elements like memory cells and gates (input, forget, and output gates) that are able to regulate the flow of information, enabling them to remember or forget specific pieces of information over long time periods. They are widely used in applications like natural language processing, speech

[3] It occurs when, through back propagation, gradients become extremely small, making it difficult for a neural network to update its weights effectively. This often leads to slow learning or completely stalled training.

recognition, and time series prediction.

- **RBF**: Radial Basis Function [19], [20] is type of neural network that uses radial basis functions as activation functions. These networks are characterized by their simplicity and ability to approximate complex functions. The RBF network consists of an input layer, a single hidden layer where the radial basis function is applied, and an output layer. Each hidden neuron computes a radial basis function based on the distance between the input and a center vector, and the results are linearly combined to produce the output. RBF networks are effective in applications like function approximation, pattern recognition, and control problems [21].

- **Gaussian Processes**: A non-parametric model used for regression and classification tasks. It defines a distribution over functions, allowing for flexible modeling of uncertainties. Each function is modeled as a collection of random variables, any finite subset of which follows a multivariate Gaussian distribution. GPs rely on kernel functions to measure similarity between data points and are particularly known for providing accurate uncertainty estimates. They are commonly applied in fields such as Bayesian optimization, time-series analysis and spatial modeling [22].

Chapter 3

Test Cases

3.1 Averaged Dynamics Environment

The first test case studies the vertical one dimensional motion of a Flapping-Wing Micro Aerial Veichle (FWMAV). This enviroment, which will be referred as Averaged Dynamics Environment (ADEnv) considers the case of a cycle-averaged motion of the drone, assuming that the natural frequency of the body is much lower than the flapping-wing frequency and thus neglecting flapping wing effects [23].

A drone of mass m, modeled as a point mass, aims to fly from initial position x_0 to the target position x_{target} in the shortest possible time and then hover in that position. The episode starts from $t = 0$ and ends at $t = t_f$.

Within this approximation, the bird is subject only to air drag, gravitational force, and a thrust force that corresponds to the lift generated by its wings. In general the lift is proportional to the air velocity impacting the wing; in this case, considering the absence of wind effects, this velocity corresponds only to the flapping velocity (i.e. rate of change of the flapping angle ϕ).

The system is governed by the equation

$$m\ddot{x} = F_p - D - F_g \tag{3.1}$$

where:

$$F_p = c_p = c_p\dot{\phi}^2 \tag{3.2}$$

$$D = \frac{1}{2} S \rho \dot{x}^2 c_D(\dot{x}) \tag{3.3}$$

$$F_g = mg \tag{3.4}$$

This second-order equation can be rewritten as a first-order system:

$$\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}. \tag{3.5}$$

Here $\mathbf{x} = [x, \dot{x}]^T = [x_1, x_2]^T$ are the system states and $\mathbf{u} = [A, f]^T = [u_1, u_2]^T$ are the control inputs, where A is the flapping amplitude and f is the flapping frequency, that drive the flapping angle ϕ and thus the flapping velocity $\dot{\phi}$. The effect of the control inputs, along with the treatment of the averaged motion, is described in Section 4.2.
The system can be written as

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = f(x_1, x_2, \dot{\phi}) = \begin{bmatrix} x_2(t) \\ \frac{1}{m}(F_p(\dot{\phi}) - D(x_2) - F_g) \end{bmatrix} \tag{3.6}$$

Regarding the drag, the drone is approximated as a blunt body with two wings; therefore the drag coefficient $c_D = c_{d,body} + 2c_{d,wing}$ can be defined as:

$$c_{d,body} = \frac{24}{\mathrm{Re}_{body}} + \frac{6}{1 + \sqrt{\mathrm{Re}_{body}}} + 0.4 \tag{3.7}$$

$$c_{d,wing} = \frac{7}{\sqrt{\mathrm{Re}_{wing}}}. \tag{3.8}$$

The body and wing's Reynolds number are then defined as:

$$\mathrm{Re}_{body} = |\dot{x}| \frac{L_{body}}{\nu} \tag{3.9}$$

$$\mathrm{Re}_{wing} = |\dot{x}| \frac{S/R}{\nu}. \tag{3.10}$$

where ν is the kinematic viscosity of air, S is the wing surface and R is the wing span.

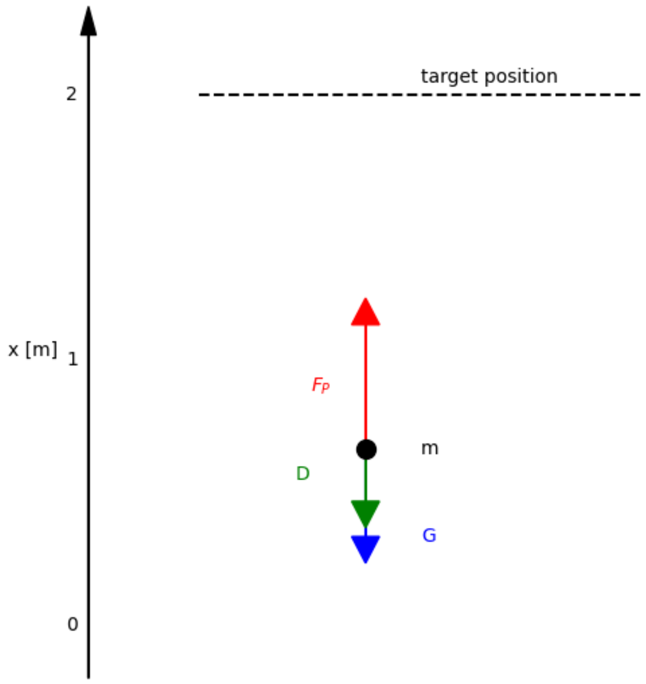


Figure 3.1: 1D motion schematics.

The problem parameters are shown in Table 3.1.

Parameter	**Value**
Mass m	$0.05\,\mathrm{kg}$
Air density ρ	$1.225\,\mathrm{kg\,m^{-3}}$
Kinematic viscosity ν	$1.460 \times 10^{-5}\,\mathrm{m^2\,s^{-1}}$
Body length L_{body}	$0.1\,\mathrm{m}$
Thrust coefficient c_p	$10^{-3.5}$
Wing area S	$5 \times 10^{-4}\,\mathrm{m^2}$
Wing span R	$0.05\,\mathrm{m}$

Table 3.1: ADEnv simulation parameters.

3.1.1 Noise

The robustness of the controllers is tested against two different types of noise: Gaussian (white) noise and OU (Ornstein–Uhlenbeck process) correlated noise. The noise $w(t)$ is added to the states during simulation as:

$$\dot{\mathbf{x}} = f(\mathbf{x} + \mathbf{w}(t), \mathbf{u}) \tag{3.11}$$

and takes the form:

- **Gaussian noise**: $w(t) = \eta(t) \sim \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(t-\mu)^2}{2\sigma^2}\right)$

where $\eta(t)$ represents a probability distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$.

- **OU noise**: $\frac{dw(t)}{dt} = -\theta w(t) + \sigma_o \eta(t)$

where $\eta(t)$ follows a Gaussian noise distribution, the parameter $\theta > 0$ is the mean reversion rate (which controls how fast the process reverts to $w(t)$) and $\sigma_o > 0$ is the volatility, which scales the magnitude of the noise.

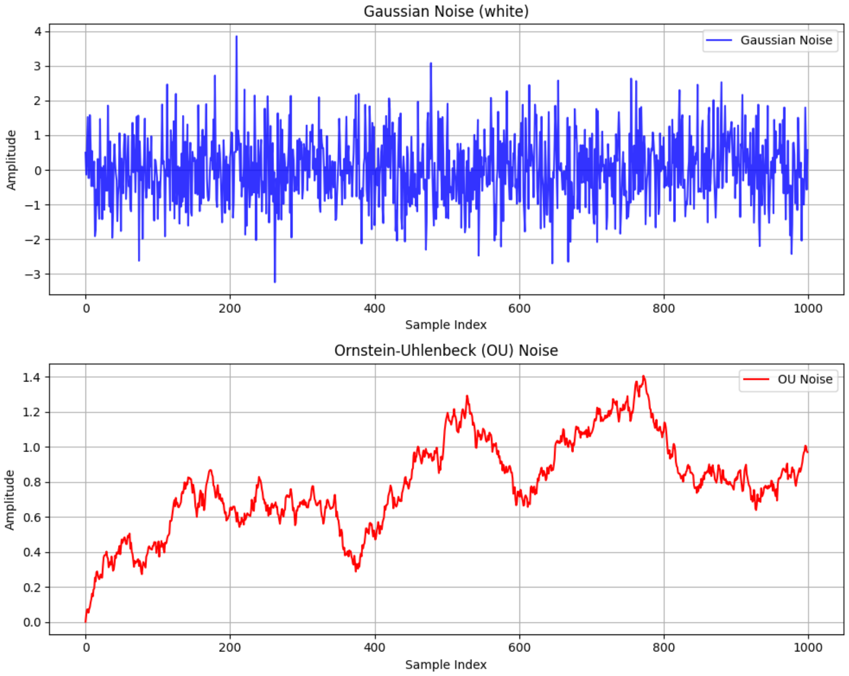


Figure 3.2: Gaussian noise (top) and OU noise (bottom).

3.2 Full Dynamics Environment

This second test case considers the control of FWMAV which seeks to move from one position to another in one dimensional motion and comes from the work of Poletti et al. [24]. Unlike the previous case, this model accounts for the full flapping-wing dynamics, rather than using an averaged-motion approximation, and will be therefore referred as Full Dynamics Environment or FDEnv. Considering the complexity of developing model-based controllers for such dynamics, this test case is addressed exclusively by using a model-free control approach, as detailed in Section 5.2.2.
The drone, initially at $\mathbf{x}_0 = [x_0, \dot{x}_0] = [0{,}0]$ is requested to move to $\mathbf{x}_{target} = [x_{target}, \dot{x}_{target}] = [5, 0]$ as fast as possible, and then hover in that position by continuously adapting its flapping wing motion.

3.2.1 Flapping motion modeling

The 1D trajectory is characterized by the vector $\mathbf{x}$ defined in the ground frame (x, y, z). A moving frame (the body frame) (x', y', z') is attached to the body with no rotational freedom. Then, three Euler angles (β_w, ϕ, α) and three reference frames identify the wing kinematics (Figure 3.3 and Figure 3.4). The wings always flap symmetrically, so the position of one wing is defined by the other one. The body frame is then pitched by a stroke plane angle $\beta_w(t)$ along y' to define the stroke plane frame (x'', y'', z'').

The wing tips flap in the stroke plane (x'', y''), which flaps along the normal z''. The third frame (flapping frame (x''', y''', z''')) follows the rotational motion and identifies the flapping angle $\phi(t)$ located between the wing symmetry axis y''' and the lateral direction y'' as in Figure (3.4). The last frame (ξ, γ, ζ) is the wing-fixed one that results from the pitching rotation along the wing symmetry axis. Then the pitching angle $\alpha(t)$ is defined between the ξ chord-normal direction and the stroke plane axis x'''. A more comprehensive treatment of the flapping-wing kinematics is available in [25].

The angles ϕ and α are parameterized by the harmonic functions:

$$\phi(t) = A_\phi cos(2\pi f t), \quad \alpha(t) = A_\alpha sin(2\pi f t), \tag{3.12}$$

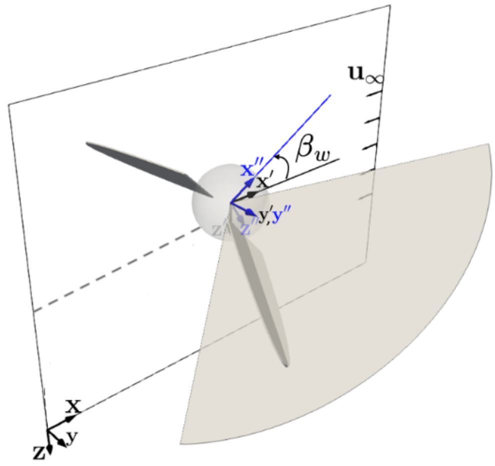


Figure 3.3: Drone schematic going from x_0 to the final position x_{target}. Reproduced from Poletti et al. [24]

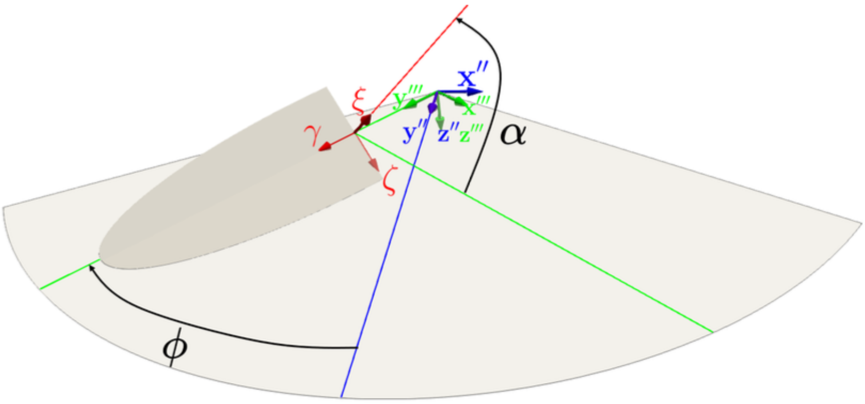


Figure 3.4: Stroke plane that contains three frames (stroke frame (x', y', z')), flapping plane (x'', y'', z'') and the wing-fixed frame (ξ, γ, ζ)), from Poletti et al. [24]

where at beginning of the stroke ($t = 0$) the flapping amplitude is maximal and the stroke plane is perpendicular to the chord plane ($\phi(t) = A_\phi$ and $\alpha(t) = 0°$). It is assumed that the pitching amplitude A_α is fixed, as well as the stroke plane angle β_w, since the motion is one dimensional.
As in the previous test case, the flapping amplitude A_ϕ and the flapping

frequency f are selected as control parameters; while in reinforcement learning terminology the control input equivalent is called action and would be denoted by $\boldsymbol{a} = [A_\phi, f]$, we continue to use $\boldsymbol{u} = [A_\phi, f]$ for consistency with the previous test case.

As mentioned before, in this test case the action is generated by a model free controller trained using a reinforcement learning algorithm (see Section 5.2.2). To make the training process more stable and efficient, it is necessary to normalize both the states (positions) and control actions to specified domains. In this implementation, Min-Max normalization is used for both states and actions:

$$\text{normalize} : [min, max] \rightarrow [a, b] \tag{3.13}$$

$$\text{denormalize} : [a, b] \rightarrow [min, max]. \tag{3.14}$$

For instance the control action $\boldsymbol{u} = [A_\phi, f]$ is produced by the policy $\boldsymbol{\pi}$, which outputs values in the normalized range $[-1, 1]$; then, before being applied to the environment, this output is denormalized into the original control range. Thus, the Min-Max functions take the form:

$$\text{normalize}(u, u_{min}, u_{max}, -1, 1) = 2\frac{u - u_{min}}{u_{max} - u_{min}} + 1 \tag{3.15}$$

$$\text{denormalize}(u, u_{min}, u_{max}, -1, 1) = \frac{1}{2}(u + 1)(u_{max} - u_{min}) + u_{min}. \tag{3.16}$$

Since the policy is stochastic (see Section 5.2.2), the output could exceed the $[-1,1]$ bounds. Therefore, to ensure valid control action, the action vector is then clipped:

$$\boldsymbol{u} = \begin{cases} clip(A_\phi, A_{\phi,min}, A_{\phi,max}) \\ clip(f, f_{min}, f_{max}) \end{cases} \tag{3.17}$$

The state variables are also normalized, and the error state $\mathbf{e} = [e, \dot{e}]$ reads:

$$\frac{x - x_{target}}{x_{target}} = e \tag{3.18}$$

$$\frac{\dot{x} - \dot{x}_{target}}{\dot{x}_{val}} = \dot{e} \tag{3.19}$$

where $\dot{x}_{val} = 5$ is a value selected for normalization, corresponding to half of the maximum velocity reached during simulations. Normalizing $\mathbf{x}_0$ and $\mathbf{x}_{target}$, we get $\mathbf{e}_0 = [-1, 0]$ and $\mathbf{e}_{target} = [0, 0]$.

We consider a drone with mass $m_b = 3$g, while the stroke plane angle β_w is fixed at 0°. The flapping amplitude is bounded within the range $[A_{\phi,min}, A_{\phi,max}] = [10°, 88°]$ while the frequency varies in $[f_{min}, f_{,max}] = [10 \text{ Hz}, 40 \text{ Hz}]$.
The drone's wings are assumed to be rigid and semi-elliptical, with a span $R = 0.05$ m and a mean chord $\bar{c}$= 0.01 m. The wing roots are offset by $R_0 = 0.0225$ m from the body's barycenter, which is also the center of rotation of the wings. Their motion is decoupled from the body, which means that the body motion is not influenced by the wing's inertia. This assumption simplifies the body dynamics, which is reduced to a function of the aerodynamic forces produced by the wings and the gravitational force. It is assumed that the forces act on the body's barycenter, treated as a material point as in the previous test case. Considering the one dimensional motion, the force balance gives again

$$m_b \ddot{x} = F_w - m_b g - D_b \tag{3.20}$$

where the subscript w refers to wing related quantities and the subscript b refers to body quantities. The aerodynamic force produced by the flapping wings is F_w and D_b is the magnitude of the drag force acting on the body, computed as

$$D_b = \frac{1}{2} \rho S_b C_{D,b} \dot{x}_b^2 \tag{3.21}$$

where ρ is the air density, $S_b = 0.0005 m^2$ is the body's cross-sectional area, $C_{D,b} = 1$ is the body's drag coefficient, and $\dot{x}$ is the relative velocity between the drone body and the wind.
To compute the wing force, the semi-empirical quasi-steady approach of Lee et al. [26] is followed. This approach provides a reasonable balance between accuracy and computational cost for the case of smooth flapping kinematics in near hovering conditions. The approach is based on the Blade Element Momentum theory (BEMT), which approximates the wings as blades and partitions them into infinitesimal elements (Figure 3.5). An

infinitesimal aerodynamic force is produced by each element, then this force is integrated along the span to get the total lift and drag: for smooth flapping motions, the flapping-generated forces dominate all the other physical effects. Hence, the lift L_w and drag D_w expressions are

$$L_w(t) = \frac{1}{2}\rho C_{L,w}(\alpha)\int_{\Delta R}^{R} U_w^2(t,r)c(r)dr \tag{3.22}$$

$$D_w(t) = \frac{1}{2}\rho C_{D,w}(\alpha)\int_{\Delta R}^{R} U_w^2(t,r)c(r)dr \tag{3.23}$$

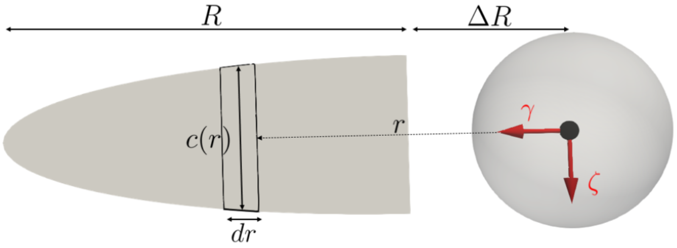


Figure 3.5: Illustration of a semi-elliptical wing, from Poletti et al. [24]

where $C_{D,w}$ and $C_{L,w}$ are the drag and lift coefficient of the wing and U_w is the magnitude of the relative velocity between the wing, the body and the wind. This velocity $\mathbf{U}_w(r,t,\mathbf{u}) = [\mathbf{U}_{w,x}, \mathbf{U}_{w,y}, \mathbf{U}_{w,z}]$ depends on the span-wise coordinate r of the wing section (see Figure 3.5) and is computed as

$$\mathbf{U}_w = \left(\mathbf{R}_2(\alpha)\begin{bmatrix}0\\0\\\dot{\phi}\end{bmatrix}\right) \times \begin{bmatrix}0\\r\\0\end{bmatrix} + \mathbf{R}_2(\alpha)\mathbf{R}_3(\phi)\mathbf{R}_2(\beta_w)\begin{bmatrix}\dot{x}\\0\\0\end{bmatrix} \tag{3.24}$$

Here the first term is the wing's linear flapping velocity, while the second term accounts for the velocity of the body. These terms are then projected in the wing frame using the rotation matrices:

$$\mathbf{R}_2(x) = \begin{bmatrix}\cos(x) & 0 & \sin(x)\\ 0 & 1 & 0\\ -\sin(x) & 0 & \cos(x)\end{bmatrix} \quad \mathbf{R}_3(x) = \begin{bmatrix}\cos(x) & \sin(x) & 0\\ -\sin(x) & \cos(x) & 0\\ 0 & 0 & 1\end{bmatrix} \tag{3.25}$$

where $\mathbf{R}_2(\beta_w)$ transform velocities from (x', y', z') to (x'', y'', z'') frame, $\mathbf{R}_3(\phi)$ from (x'', y'', z'') to (x''', y''', z'''), $\mathbf{R}_2(\alpha)$ from (x''', y''', z''') to (ξ, γ, ζ).

The coefficients $C_{L,w}$ and $C_{D,w}$ are modeled similarly to Lee et al. [26]:

$$C_{L,w} = a\sin(2\alpha_e) \tag{3.26}$$

$$C_{D,w} = b + c(1 - \cos(2\alpha_e)) \tag{3.27}$$

where the coefficient $[a, b, c] = [1.71, 0.043, 1.595]$, as in Poletti et al. [24] and Lee et al. [26] for semi-elliptical wings. This parameterisation models the influence of the Leading Edge Vortex (LEV), generated at a high pitching angle due to flow separation and stably attached on the suction side (as can be seen in Sane [27]). Then the force coefficients $C_{D,w}$ and $C_{L,w}$ depend on the effective angle of attack defined as

$$\alpha_e = \arccos(\frac{U_{w,z}}{||\mathbf{U}_w||_2}) \tag{3.28}$$

between the relative velocity of the wing U_w and the chord direction ζ. The forces in equations (3.22) (3.23) are computed in the reference wind frame (ξ', γ', ζ'), which rotates the wing frame by α_e along γ. The drag D_w is aligned with ξ' and perpendicular to the lift (Figure 3.6). Finally the forces are transformed into the wing and then into the body frame and the force F_w is retrieved.

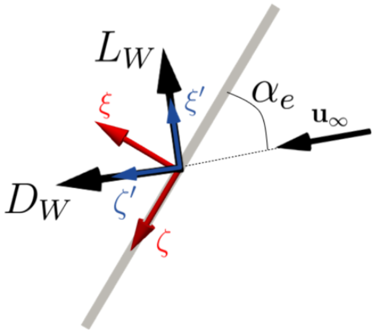


Figure 3.6: α_e on a chord section of the wing, taken from Poletti et al. [24]

The problem parameters are summarized in Table 3.2

Parameter	**Value**
Mass m	$0.003\,\mathrm{kg}$
Air density ρ	$1.225\,\mathrm{kg\,m^{-3}}$
Body drag coefficient C_{D_b}	1.0
Body cross-section area S_b	$0.0005\,\mathrm{m^2}$
Stroke plane angle β_w	$0°$
$[A_{\phi,\mathrm{min}}, A_{\phi,\mathrm{max}}]$	$[10°, 88°]$
$[f_{\mathrm{min}}, f_{\mathrm{max}}]$	$[10\,\mathrm{Hz}, 40\,\mathrm{Hz}]$
Wing span R	$0.05\,\mathrm{m}$
Mean chord $\bar{c}$	$0.01\,\mathrm{m}$
Wing root offset R_0	$0.0225\,\mathrm{m}$

Table 3.2: FDEnv simulation parameters.

Chapter 4

Model Based control

For the model based approach, in order to perform optimal control, a time-discrete Linear Quadratic Regulator (LQR), initially introduced by Rudolf Kalman in 1960 [28], will be used. This theory employs a quadratic cost function and quadratic terms for state and control variables. The behavior of the controller is shaped by weighting matrices, allowing for the prioritization of states and control inputs to achieve the desired performance. To compute the controller, the Riccati equation is solved, and then different techniques are applied to obtain the necessary control gains for the controller design. The LQR demonstrates its ability to stabilize unstable systems by minimizing a cost function. However, variations in system parameters can affect the control gains, prompting the use of optimization techniques to adjust them for stability and optimal performance despite changes.

4.1 LQR

Consider a discrete-time linear system in state-space form

$$\mathbf{x}_{\mathrm{k+1}} = \mathbf{A}\mathbf{x}_{\mathrm{k}} + \mathbf{B}\mathbf{u}_{\mathrm{k}} \tag{4.1}$$

with states $\mathbf{x}_k$, controls $\mathbf{u}_k$ and weight matrices $\mathbf{Q}$ and $\mathbf{R}$. The finite-horizon cost function used to stabilize the system at the origin is defined as:

$$J = \sum_{k=t_0}^{t_f} \mathbf{x}_k^T\mathbf{Q}\mathbf{x}_k + \mathbf{u}_k^T\mathbf{R}\mathbf{u}_k, \quad \mathbf{Q} = \mathbf{Q}^T \succeq \mathbf{0}, \mathbf{R} = \mathbf{R}^T \succ 0 \tag{4.2}$$

where the first term of the sum penalizes deviation from the desired state, while the second term penalizes control effort. Following the Bellman optimality principle, the goal is to minimize the cost-to-go given by

$$J(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}) = J_{k-1} = \mathbf{x}_k^T \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k + J(\mathbf{x}_k, \mathbf{u}_k) \tag{4.3}$$

The optimal cost-to-go has to be quadratic, and it's chosen to be of the form

$$J(\mathbf{x}_k) = J_k = \mathbf{x}_k^T \mathbf{P}_k \mathbf{x}_k, \quad \mathbf{P}_k = \mathbf{P}_k^T \succ 0. \tag{4.4}$$

Plugging (4.4) back into equation (4.3) and considering (4.1) we get

$$J_{k-1} = \mathbf{x}_k^T \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k + \mathbf{x}_k^T \mathbf{A}^T \mathbf{P}_k \mathbf{A} \mathbf{x}_k + \mathbf{u}_k^T \mathbf{B}^T \mathbf{P}_k \mathbf{B} \mathbf{u}_k + 2\mathbf{x}_k^T \mathbf{A}^T \mathbf{P}_k \mathbf{B} \mathbf{u}_k =$$

$$= \mathbf{x}_{k-1}^T \mathbf{P}_{k-1} \mathbf{x}_{k-1}. \tag{4.5}$$

To find the optimal control $\mathbf{u}_k^*$ we minimize (4.5) with respect to $\mathbf{u}_k$: we take its derivative with respect to $\mathbf{u}_k$ and we set it to zero

$$2\mathbf{R}\mathbf{u}_k + 2\mathbf{B}^T \mathbf{P}_k \mathbf{A} \mathbf{x}_k + 2\mathbf{B}^T \mathbf{P}_k \mathbf{B} \mathbf{u}_k = 0 \tag{4.6}$$

and we solve it for $\mathbf{u}_k$, obtaining the optimal closed loop feedback control law:

$$\mathbf{u}_k^* = -(\mathbf{R} + \mathbf{B}^T \mathbf{P}_k \mathbf{B})^{-1} \mathbf{B}^T \mathbf{P}_k \mathbf{A} \mathbf{x}_k = -\mathbf{K} \mathbf{x}_k \tag{4.7}$$

where K is the gain matrix. Substituting (4.7) into (4.5) we obtain the discrete time Riccati equation (DARE):

$$\mathbf{P}_{k-1} = \mathbf{Q} + \mathbf{A}^T \mathbf{P}_k \mathbf{A} - (\mathbf{A}^T \mathbf{P}_k \mathbf{B})(\mathbf{R} + \mathbf{B}^T \mathbf{P}_k \mathbf{B})^{-1} (\mathbf{B}^T \mathbf{P}_k \mathbf{A}), \quad \mathbf{P}_{k_f} = 0 \tag{4.8}$$

where the terminal condition $\mathbf{P}_{k_f} = 0$ is assigned under the assumption that there is no terminal cost (i.e. $J_{k_f} = 0$). By solving (4.8) for $\mathbf{P}_k$, it is possible to compute the gain matrix needed for optimal control law $\mathbf{u}_k^*$.

4.1.1 Weight Matrices

One of the major hurdles of LQR control is the selection of weighing matrices Q and R. The conventional approach deals with a trial and error approach: for example, by assigning large values to R matrix we can

penalize the control effort heavily. Similarly, assigning large values to Q matrix penalizes the deviation from reference states. To choose Q and R, the simplest choice is $\mathbf{Q} = \mathbf{I}$ and $\mathbf{R} = \rho\mathbf{I}$ and then to vary ρ to get something that has good response.
Alternatively, one can use *diagonal weights*:

$$\mathbf{Q} = \begin{bmatrix} q_1 & & \\ & \ddots & \\ & & q_n \end{bmatrix} \qquad \mathbf{R} = \rho \begin{bmatrix} r_1 & & \\ & \ddots & \\ & & r_n \end{bmatrix} \tag{4.9}$$

and then choosing each term q_i, r_i as a "measure of badness". The underlying principle is to choose each weight such that $q_i x_i^2 = 1$ when x_i equals its acceptable error: this way, all terms in the cost function are normalized, reflecting equivalent penalty contributions (the same principle applies to $r_i u_i^2 = 1$ for the control effort).

For instance, considering a system similar to (4.1), we aim to impose the same relative effort to control different state variables. Considering the ADEnv, where the state variable x_1 represents a position in meters and x_2 a velocity in millimeters per second, and assuming acceptable errors of 1 cm and 1 mm/s for x_1 and x_2 respectively, we get:

$$\text{acceptable error of 1 cm} \quad \rightarrow \quad q_1 = \tfrac{1}{(0.01)^2}\ \text{m}^{-2}$$

$$\text{acceptable error of 1 mm/s} \quad \rightarrow \quad q_2 = \tfrac{1}{(0.001)^2}\ (\text{m/s})^{-2}$$

This ensures each term contributes equally when the state reaches its respective acceptable error. Moreover, the tuning factor ρ can be used to adjust the control/state cost balance.
Finally, with the advent of machine learning optimization techniques, hyper parameters tuning became more effective thanks to algorithms like Particle Swarm Optimization [29] and Bayesian Optimization [30].

4.2 Trajectory tracking

Suppose we are given a system $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ and a feasible trajectory $(\mathbf{x}_*, \mathbf{u}_*)$. The goal is to design a controller of the form $\mathbf{u} = \mathbf{u}(\mathbf{x}, \mathbf{x}_*, \mathbf{u}_*)$

such that $\lim_{t\to\infty}(\mathbf{x}-\mathbf{x}_*)=0$. This is known as the trajectory tracking problem; this trajectory $(\mathbf{x}_*, \mathbf{u}_*)$ can also consist of a single point (such as in this work) and it's called *operating point*.

The tracking problem is very well suited when we are dealing with a non linear system such as the first environment: in fact, in order to use LQR theory, it's necessary to reduce the non linear system in (3.1) to a linear one of the form $\mathbf{x}_{\mathrm{k}+1} = \mathbf{A}\mathbf{x}_{\mathrm{k}} + \mathbf{B}\mathbf{u}_{\mathrm{k}}$.

Assuming to have an operating point $(\mathbf{x}_*, \mathbf{u}_*)$, first we define a local relative coordinate system To design the controller, it is necessary to construct the error system by defining a local relative coordinate system:

$$\Delta\mathbf{x} = \mathbf{x}-\mathbf{x}_*, \quad \Delta\mathbf{u} = \mathbf{u}-\mathbf{u}_* \tag{4.10}$$

and therefore $\Delta\dot{\mathbf{x}} = \dot{\mathbf{x}} - \dot{\mathbf{x}}_* = f(\mathbf{x},\mathbf{u}) - f(\mathbf{x}_*,\mathbf{u}_*)$. Now, assuming that $\bar{\mathbf{x}}$ is small (so the controller is doing a good job), it's possible to linearize the system around $(\mathbf{x}_*, \mathbf{u}_*)$ using first-order Taylor expansion:

$$\begin{aligned}\Delta\dot{\mathbf{x}} \approx f(\mathbf{x}_*,\mathbf{u}_*) + \frac{\partial f(\mathbf{x}_*,\mathbf{u}_*)}{\partial \mathbf{x}}(\mathbf{x}-\mathbf{x}_*) + \frac{\partial f(\mathbf{x}_*,\mathbf{u}_*)}{\partial \mathbf{u}}(\mathbf{u}-\mathbf{u}_*) - f(\mathbf{x}_*,\mathbf{u}_*) = \\ = \tilde{\mathbf{A}}\Delta\mathbf{x} + \tilde{\mathbf{B}}\Delta\mathbf{u} \qquad (4.11)\end{aligned}$$

In the most general case, by designing a state feedback controller $\mathbf{K}(\mathbf{x}_*)$ for each $\mathbf{x}_*$ ($\mathbf{K}$ is a function of the trajectory) we can regulate the system using the feedback

$$\Delta\mathbf{u}_k = -\mathbf{K}(\mathbf{x}_*)\Delta\mathbf{x}_k \quad \Rightarrow \quad \mathbf{u}_k - \mathbf{u}_* = -\mathbf{K}(\mathbf{x}_*)(\mathbf{x}_k - \mathbf{x}_*) \tag{4.12}$$

This form of controller is called a gain scheduled linear controller with feed forward $\mathbf{u}_*$.

If we consider a linear system and an operating point as trajectory, the matrix $\mathbf{K}$ becomes constant. Considering the ADEnv, by choosing as operating point the hovering position $\mathbf{x}_* = \mathbf{x}_{target} = [2,0]$, $\mathbf{u}_* = [A_*, f_*]$ we get

$$\begin{aligned}\tilde{\mathbf{A}} &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}_{x_*,u_*} = \begin{bmatrix} 0 & 1 \\ 0 & \frac{\partial D}{\partial x_2} \end{bmatrix}_{x_*,u_*} \\ \tilde{\mathbf{B}} &= \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} \end{bmatrix}_{x_*,u_*} = \begin{bmatrix} 0 & 0 \\ \frac{\partial \bar{F}_p}{\partial u_1} & \frac{\partial \bar{F}_p}{\partial u_2} \end{bmatrix}_{x_*,u_*}\end{aligned} \tag{4.13}$$

with

$$\bar{F}_p = c_p u_1^2 u_2^2 = c_p A^2 f^2 \tag{4.14}$$

$$\frac{\partial \bar{F}_p}{\partial u_1} = \frac{\partial \bar{F}_p}{\partial A} = 2c_p A f^2 \tag{4.15}$$

$$\frac{\partial \bar{F}_p}{\partial u_2} = \frac{\partial \bar{F}_p}{\partial f} = 2c_p A^2 f \tag{4.16}$$

and

$$D = \frac{1}{2} S \rho \dot{x}^2 c_D(\dot{x}) \tag{4.17}$$

$$\frac{\partial D}{\partial x_2} = \frac{\partial D}{\partial \dot{x}} = \frac{1}{2} S \rho \left(2\dot{x} c_D + \dot{x}^2 \frac{\partial c_D}{\partial \dot{x}} \right) \tag{4.18}$$

$$\frac{\partial c_D}{\partial \dot{x}} = c_{d,body} = \frac{24}{\mathrm{Re}_{body}} + \frac{6}{1 + \sqrt{\mathrm{Re}_{body}}} + 0.4 \tag{4.19}$$

$$D = \frac{1}{2} S \rho \dot{x}^2 c_D(\dot{x})$$

$c_D = (c_{d,body} + 2c_{d,wing})$ reads as follow:

$$c_{d,body} = \frac{24}{\mathrm{Re}_{body}} + \frac{6}{1 + \sqrt{\mathrm{Re}_{body}}} + 0.4$$

$$c_{d,wing} = \frac{7}{\sqrt{\mathrm{Re}_{wing}}}$$

where

$$\mathrm{Re}_{body} = |\dot{x}| \frac{L_{body}}{\nu} \tag{4.20}$$

$$\mathrm{Re}_{wing} = |\dot{x}| \frac{S_{wing}/R_{wing}}{\nu}. \tag{4.21}$$

To obtain the discrete-time matrices we compute

$$\begin{aligned} \mathbf{A} &= \tilde{\mathbf{A}}dt + \mathbf{I} \\ \mathbf{B} &= \tilde{\mathbf{B}}dt \end{aligned} \tag{4.22}$$

and the resulting controller reads

$$\mathbf{u}_k^* = \mathbf{u}_* - \mathbf{K}(\mathbf{x}_k - \mathbf{x}_*). \tag{4.23}$$

Chapter 5

Model Free control

5.1 Policy gradient

As already said, the goal of a reinforcement learning agent is to maximise expected reward over it's lifetime. For a random variable x, the *expected value* defines what is expected to happen on average; thus, the expectation operator $\mathbb{E}$ is introduced:

- discrete case: $\mathbb{E}[x] = \sum_{i=0}^{N} x_i \cdot p_i$
- continuous case: $\mathbb{E}(x) = \int x \cdot p(x) dx$

which is the average of all possible values that x can assume, weighted by its probability p_i. For the continuous case, the discrete probability p_i is replaced by a probability density function $p(x)$.

The trajectory $\tau = \langle s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_T \rangle$ is defined by the sequence of states, actions, and rewards collected during the interactions between the agent and the environment. The agent selects actions according to a policy $\pi(a_t|s_t)$, and the environment responds by transitioning to a new state s_{t+1} and giving a reward r_t; each state s_{t+1} depends on the previous state s_t and action a_t through the environment's transition dynamics. Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon[1] undiscounted return of the policy over τ

[1] the episode ends at T, thus each trajectory τ is a finite set

$$J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}[R(\tau)] = \int_\tau R(\tau)\cdot p(\tau|\pi_\theta) \tag{5.1}$$

that is, for continuous policies, the reward expectation over the trajectories. When dealing with discrete-sampled environment (but still continuous policies), the expectation becomes:

$$\underset{\tau\sim\pi_\theta}{\mathbb{E}}[R(\tau)] = \frac{1}{N}\sum_{i=1}^{N} R(\tau_i)\cdot p(\tau_i|\pi_\theta) \tag{5.2}$$

We want to find the parameters that maximise this objective, and hence find an optimal parameterisation for π.

5.1.1 Vanilla Policy Gradient

Considering a stochastic policy $\pi_\theta(a|s)$, the key idea underlying policy gradients is to modify the parameters θ in order to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy. To do so, we find the parameters θ that maximise the objective J trough gradient ascent; thus it's necessary to have some sort of expression of the gradient $\nabla_\theta J$ with respect to the parameters. This expression is derived analytically in the appendix (A.1) and reads:

$$\nabla_\theta J(\pi_\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau) R(\tau). \tag{5.3}$$

The "vanilla" policy gradient, also called REINFORCE (REward Increment = Nonnegative Factor x Offset Reinforcement x Characteristic Eligibility [31]), works by updating policy parameters via gradient ascent based on the policy performance:

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J \tag{5.4}$$

Equations (5.3) and (5.4) tell that the policy parameters θ are shifted in the average direction that increases the log probability of taking a trajectory proportionally to how large was the reward $r(\tau)$ along that trajectory (times a learning rate α). This way the bigger R for some τ

was, the more likely those trajectories are to be chosen again.

5.1.2 Exploration vs Exploitation

REINFORCE trains in an on-policy way a stochastic policy, meaning that it explores by sampling actions according to the latest version of its policy. The randomness in its action selection depends on both initial conditions and the training process. Over the course of training, the policy usually becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found (even if it may cause to get trapped in local optima).
However, this algorithm is subject to large variance and doesn't work in practice. Considering our test cases (see Section 5.2.2), the total reward accumulated over each trajectory is always negative. As a result, the log-probability of any trajectory, regardless of whether it was better or worse, will be reduced by the updates: this impacts learning, as the algorithm cannot distinguish between good and bad behaviors by only accounting for absolute returns.
To avoid this, a baseline $b(s)$ can provide a reference value that allows the algorithm to assess how good a particular trajectory is relative to others, rather than in absolute terms:

$$\nabla_\theta J(\pi_\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau)[R(\tau) - b(s)] \tag{5.5}$$

The baseline $b(s)$ does not depend on the current trajectory (i.e. it depends only on states) and weights the influence of the trajectory on the gradient, centering the distribution of rewards across trajectories around zero. A standard baseline could be the average over trajectories:

$$b(s) = \frac{1}{N}\sum_{i=1}^{N} r(\tau_i)$$

Even though we adjust the objective by subtracting a baseline, the expected value of the gradient is not affected. This is because the baseline does not depend on the action (only on the state) and, as shown in (A.2), its expected contribution to the gradient is zero. This way, the baseline helps reduce variance without altering the direction of the expected gradient.

5.1.3 Causality

It is important to consider which rewards an action should be held accountable for: intuitively, a reward obtained prior to the execution of a given action cannot be attributed to that action. Therefore, the evaluation of an action's effectiveness should only consider the rewards obtained from the time the action is taken onward. Rewards received prior to the action are not attributable to it and should therefore not influence its likelihood of being selected in the future.
This leads to a modified version of the policy gradient, where $\nabla_\theta J(\pi_\theta)$ is weighted by the return from the current time step onward:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \log \pi_\theta(a_t^i | s_t^i) \left[\sum_{t'=t}^{T} r(a_{t'}^i, s_{t'}^i) \right] \tag{5.6}$$

where i is the subscript for the trajectories, t is the timestep in an episode and the sum over t' is to take in account only future rewards.
This sum of rewards could be also discounted of a factor γ, thus obtaining the q-value

$$Q_t = Q(a_t, s_t) = \sum_{t'=t}^{T} \gamma^{t'} r(a_{t'}, s_{t'}) \tag{5.7}$$

5.1.4 Advantage Function

Many algorithms used today (PPO, SAC, A2C,...) share the idea of dividing into value and advantage, forming the family of actor-critic algorithms. The advantage function is basically an estimate of the relative value for a selected action and tries evaluate whether a specific action of the agent is better or worse than some other possible action in a given state.
Remembering the notions of value function V^π and action-value function Q^π for a policy π, we denote the advantage for the policy π_θ as A^{π_θ}: this advantage can be in general defined as $A(s, a) = Q(s, a) - V(s)$, where Q

corresponds to the discounted sum of rewards (the total weighted reward for the completion of an episode as in (5.7)) and V is the baseline estimate. The relation between V and Q is:

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma V(s_{t+1}) \tag{5.8}$$

If the output of this function is positive ($A > 0$), it means that the sampled return of the action is better than the expected return from experience, so the possibilities of selecting that specific action will increase through the update rule (for instance Eq (5.5)); if $A < 0$, the actual return is worse and the probability of selecting that action will diminish.

The advantage function is calculated after the completion of each episode by recording its outcome to evaluate Q (5.7) and V from the value function. Once Q and V are computed, the advantage function can be calculated by subtracting the baseline estimate V from the actual discounted return. However, the baseline corresponds to the value function that outputs the expected discounted sum of an episode starting from the current state: because there aren't rules or models to derive it, it has to be modeled as well, and thus learned, with some function function approximator, typically via regression on mean-squared error (5.13). If the learning is done on-policy, such as in PPO the baseline estimate will be noisy (with some variance) as the policy function itself.

It's important to note that there are different ways to calculate the advantage other than the action-value Q function such as Direct Advantage Estimator (DAE) [32], Generalized Advantage Estimator (GAE) [33] and others. Typically, policy gradient implementations compute advantages estimates using the infinite-horizon discounted return, despite otherwise using the undiscounted finite-horizon policy gradient equation.

5.1.5 Proximal Policy Optimisation

Actor-critic methods still face two drawbacks:

- While reducing the variance with respect to vanilla policy gradient, the learning process is still ineffective: depending on the learning rates, it's usually either too slow or it misses the best policies and trajectories.

- On-policy algorithm are data-inefficient: the data gathered is lost every policy update, resulting in slow learning.

In order to improve training stability, a family of algorithms called Trust Region Methods was developed (such as Trust Region Policy Optimisation, TRPO [34]), where constraints are imposed on how much the policy can deviate from the previous one each time is updated. Belonging to this family, Proximal Policy Optimization (PPO, Schulman et al. [16]) was introduced by OpenAI in 2017 and was designed to keep the stability and performance benefits of TRPO while simplifying its implementation and reducing computational cost, emerging as one of the most widely used RL algorithms.

At the core of PPO is the idea of constraining policy updates using a clipped objective function, without explicitly measuring the distance between policies. Instead of relying on computationally expensive divergence metrics (such as in TRPO), PPO directly limits the change in action probabilities using a *policy ratio*, defined as:

$$\tilde{r}_\theta = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta,old}(a_t|s_t)} \tag{5.9}$$

which compares the likelihood of taking action a_t in state s_t under the new and old policies. Then, this ratio is kept in a certain proximal range defined by a parameter ϵ: to ensure that $\tilde{r}_\theta$ does not deviate too far from 1 (thus preventing large and potentially harmful updates), PPO clips it within the bounded range $[1-\epsilon, 1+\epsilon]$. The goal is to avoid large step in the parameter space that may overshoot regions of quick improvement and enter areas where the policy becomes degenerate: since the policy is used to sample training data (i.e. on-policy learning), recovery from such regions could be difficult. Instead, taking smaller steps increases the chances that the policy continues to improve monotonically.
To implement this, two surrogate objective functions that approximate the policy improvement are introduced:

$$\text{surrogate 1:} \qquad J_1^{surr} = \tilde{r}_\theta A^{\pi_\theta}(s_t, a_t) \tag{5.10}$$

$$\text{surrogate 2:} \qquad J_2^{surr} = \text{clip}(\tilde{r}_\theta, 1-\epsilon, 1+\epsilon) A^{\pi_\theta}(s_t, a_t). \tag{5.11}$$

These represent the product between the advantage A^{π_θ} and either the policy ratio $\tilde{r}_\theta$ or its clipped value. By comparing the updated policy to the previous one, the clipped objective $J^{CLIP}(\theta)$ accounts the minimum between these two surrogates and corresponds to its expected value:

$$J^{CLIP}(\theta) = \mathbb{E}[\min(\tilde{r}_\theta A^{\pi_\theta}(s_t, a_t), \text{clip}(\tilde{r}_\theta, 1-\epsilon, 1+\epsilon)A^{\pi_\theta}(s_t, a_t))] \quad (5.12)$$

By choosing the minimum value between the two surrogates, we are taking small steps only when the advantage suggests improvement: if we are fixing an issue with the policy where the ratio is smaller than 1 (which means that the old policy assigns a greater probability to a beneficial action than new one) the step is unbounded and we can move as far as indicated by the calculation in the negative direction.
As for the value network that models V^π, the loss function often corresponds to the mean squared error (MSE) between actual and predicted values: recalling the definition of Q_t (5.7), this function reads

$$J^V(\psi) = \sum_{t=0}^{T}(Q_t - V_t^\pi(\psi))^2. \quad (5.13)$$

During training, the overall objective function is usually the linear combination of three term: J^{CLIP} from 5.12, J^V from equation 5.13 and an entropy term S added to promote exploration (see Schulman et al. [16]).

5.2 Radial Basis Function

In this work Radial Basis Function (RBF) will be used as function approximator, which takes the form of $\hat{y}(\mathbf{x}) = \sum_{i=1}^{N} \phi_i(\mathbf{x} - \mathbf{x}_i)w_i + w_0$ where ϕ is the so called radial basis function.
An approximator of this kind can be seen as a simple neural network, consisting in only one hidden layer and one output layer:

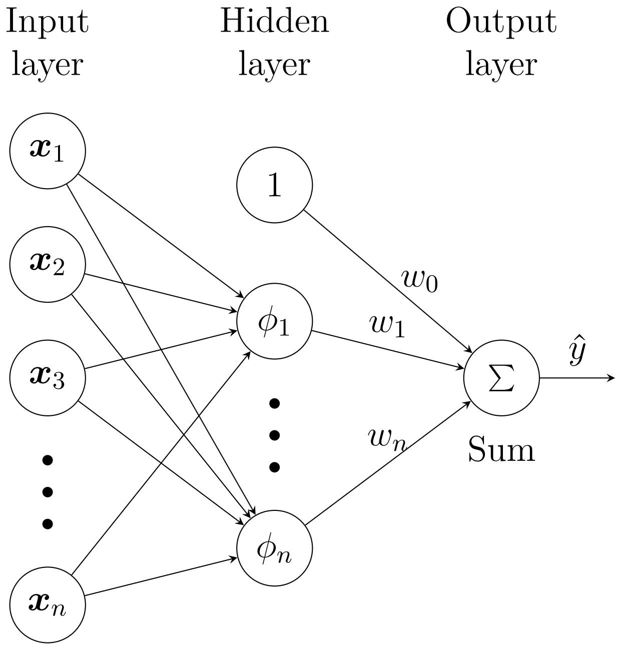


Figure 5.1: RBF neural network.

This type of function is *radial* in the sense that it depends only on the distance from the point in which is collocated, called *center*; the distance is usually Euclidean distance ($||\cdot||_2$), but other metrics can be used. The bases ϕ_k are used as collection $\{\phi_k\}_k = \mathbf{\Phi}$ forming a basis for a function space, hence the name. They are particularly useful because, other than being infinitely differentiable, they are easier to implement and understand compared to other neural architectures.
Multiple *bases* can be used, some examples are:

- $\phi(r) = \sqrt{1 + (\epsilon r)^2}$ Multiquadratic Radial Basis
- $\phi(r) = \frac{1}{1+(\epsilon r)^2}$ Inverse Radial Basis
- $\phi(r) = \frac{1}{\sqrt{1+(\epsilon r)^2}}$ Inverse Multiquadratic Radial Basis
- $\phi(r) = \exp(-\epsilon r^2)$ Gaussian Radial Basis
- $\phi(r) = \exp(-(\epsilon r)^2)((\epsilon r)^2 + 3(\epsilon r) + 3)$ Matern C4 Radial Basis

where r is the radius (i.e. $||\mathbf{x} - \mathbf{x}_c||_2$) and ϵ is a positive constant. Four of them are represented in Figure 5.2.

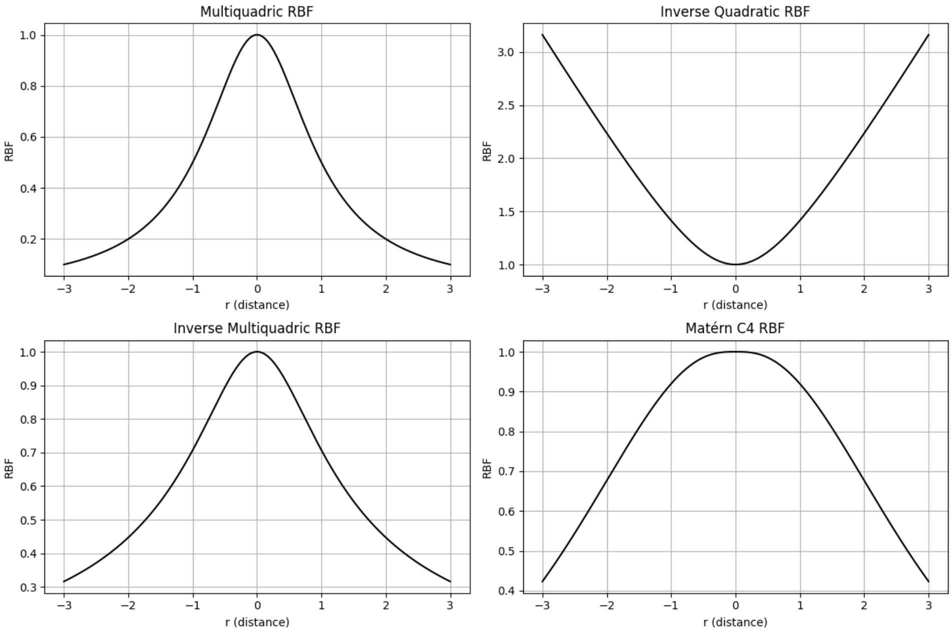


Figure 5.2: RBF with different kernels.

To model the policy in this work, the Gaussian RBF was selected. In particular, considering the bi-dimensional state space of our test cases, the basis reads

$$\phi(\mathbf{x} - \mathbf{x}_i) = \exp(\ -\boldsymbol{\epsilon}^2(\mathbf{x} - \mathbf{x}_i)^2\ ) = \mathrm{e}^{-(\ \epsilon_x^2(x-x_i)^2+\epsilon_{\dot{x}}^2(\dot{x}-\dot{x}_i)^2\ )} \tag{5.14}$$

where $\boldsymbol{\epsilon}, \mathbf{x}, \mathbf{x}_i \in \mathbb{R}^2$, $\mathbf{x} = [x, \dot{x}]^T$; $\mathbf{x}_i = [x_i, \dot{x}_i]^T$ are the centers distributed along the two dimensions and $\boldsymbol{\epsilon} = [\epsilon_x, \epsilon_{\dot{x}}]$ is the shape parameter (or sensitivity) with respect to each dimension. In Figure 5.3, it's shown the influence of ϵ since all the other parameters (centers, weights) are the same.

To define the policy, first we define the bounds for each dimension ($x_{min}, x_{max}, \dot{x}_{min}$ and $\dot{x}_{max}$). Now, considering the sets $\boldsymbol{x} = [x_{min}, ..., x_{max}] \subset \mathbb{R}^n$ and $\dot{\boldsymbol{x}} = [\dot{x}_{min}, ..., \dot{x}_{max}] \subset \mathbb{R}^m$, the elements of the product $\boldsymbol{x} \times \dot{\boldsymbol{x}} \subset \mathbb{R}^{n\times m}$ constitute the *centers* of the multidimensional RBF (represented

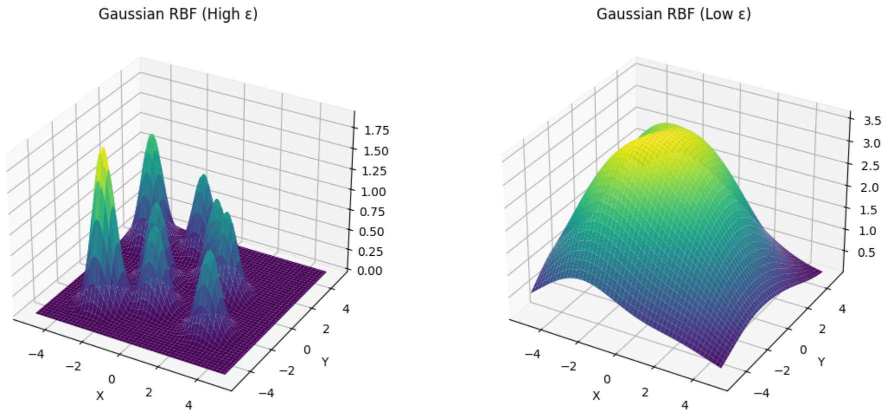


Figure 5.3: 2D gaussian RBF, influence of shape parameter ϵ.

in Figure 5.3).
The equation for the RBF network reads as equation (5.15):

$$\hat{y} = w_0 + \sum_{i=1}^{n}\sum_{j=1}^{m} \phi(\mathbf{x} - \mathbf{x}_{ij}) w_{ij} = \mathbf{\Phi}(\mathbf{x})\boldsymbol{w} \tag{5.15}$$

$$\mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} 1 & \phi(\mathbf{x}-\mathbf{x}_{11}) & \phi(\mathbf{x}-\mathbf{x}_{12}) & \phi(\mathbf{x}-\mathbf{x}_{13}) & \dots & \phi(\mathbf{x}-\mathbf{x}_{1n}) \\ 1 & \phi(\mathbf{x}-\mathbf{x}_{21}) & \phi(\mathbf{x}-\mathbf{x}_{22}) & \phi(\mathbf{x}-\mathbf{x}_{23}) & \dots & \phi(\mathbf{x}-\mathbf{x}_{2n}) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \phi(\mathbf{x}-\mathbf{x}_{m1}) & \phi(\mathbf{x}-\mathbf{x}_{m2}) & \phi(\mathbf{x}-\mathbf{x}_{m3}) & \dots & \phi(\mathbf{x}-\mathbf{x}_{mn}) \end{bmatrix}$$

$$\boldsymbol{w} = \begin{bmatrix} w_{01} & w_{02} & w_{03} & \dots & w_{0m} \\ w_{11} & w_{12} & w_{13} & \dots & w_{1m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & w_{n3} & \dots & w_{nm} \end{bmatrix}$$

$$w_0 = \sum_{i=1}^{m} w_{0i}$$

5.2.1 ADEnv - RBF Implementation

In this test case, the policy $\boldsymbol{\pi}_\theta = \boldsymbol{\pi}_\theta(\mathbf{x}) = [\pi_A(\mathbf{x}), \pi_f(\mathbf{x})]$ is represented by two separate RBF networks of the form (5.15), one for each control input (amplitude and frequency). Thus

$$\mathbf{u}(\mathbf{x}) = \boldsymbol{\pi}_\theta(\mathbf{x}). \tag{5.16}$$

The two RBFs share the same number of centers. The centers' position $\mathbf{x}_i$, sensitivities $\boldsymbol{\epsilon}$ and weights $\boldsymbol{w} \setminus \{w_0\}$ (excluding the bias term w_0) of each RBF are learnable parameters, and the bias term w_0 is fixed to stabilize the learning process. The parameters are presented in the Results section, in Table 6.1.

As mentioned before, in the averaged test case we design the model free controller by direct optimization using the Adam optimizer. This optimizer was developed by Kingma and Ba [35] and combines the benefits of momentum and adaptive learning rates. It estimates the mean and the uncentered variance of the gradients, along with bias correction terms. It also incorporates momentum by using an exponentially decaying average of past gradients, improving convergence stability and speed in high-dimensional, noisy, and sparse settings. In particular, in this work we used the PyTorch's version, which incorporates the differentiation chain rule through automatic differentiation system (Autograd), enabling stable and efficient parameter updates.

Finally, the reward function used during optimization penalizes both the distance and velocity errors from the target state and is defined as:

$$R = -\sum_{t=0}^{T} ||\Delta x||_2^2 + 0.01||\Delta \dot{x}||_2^2 \tag{5.17}$$

where the coefficient (0.01) weights the importance of the velocity penalty.

5.2.2 FDEnv - RBF Implementation

While the smooth dynamics of ADEnv allowed the use of direct optimization via Adam, the dynamics of FDEnv are significantly more complex. This environment exhibits higher nonlinearity, temporal dependencies and explicit frequency-related effects, which make it poorly suited to direct optimization methods that typically perform better in static or smooth settings. In the FDEnv, the absence of the averaging assumption exposes the flapping frequency effects, which not only increase the system's dynamical complexity but also introduce a rapidly varying control input. As

a result, the control process becomes highly dynamic and less suited to the method used in the simplified test case.
By contrast, reinforcement learning algorithms are better equipped to handle such challenges. Their ability to manage stochasticity, delayed rewards and complex time-dependent interactions makes them more suitable for environments like FDEnv, where these properties are more pronounced.

As the previous test case, the policy $\boldsymbol{\pi}_\theta = \boldsymbol{\pi}_\theta(\mathbf{x}) = [\pi_A(\mathbf{x}), \pi_f(\mathbf{x})]$ is again represented by two separate RBF networks of the form (5.15).
To assess the capabilities of this policy approximator, training was first attempted using the simplest policy gradient algorithm (REINFORCE, see Section 5.1.1). The goal was to determine whether the simplicity of the RBF architecture would allow such a basic algorithm to solve the control problem. Due to unsatisfactory results, the approach was upgraded to a minimal implementation of Proximal Policy Optimization, as described in Algorithm 1. This second method essentially corresponds to an actor-critic implementation of REINFORCE, enhanced by a clipped surrogate objective to improve training stability.
The reward function for this environment, is defined as:

$$R = -\frac{1}{T}\sum_{t=0}^{T} r_t = -\frac{1}{T}\sum_{t=0}^{T} ||e_t||_2^2 + 0.01||\dot{e}_t \cdot h(x_t)||_2^2 \tag{5.18}$$

where $h(x_t)$ is a Gaussian function with mean x_f and standard deviation 0.71

$$h(x_t) = \frac{x_{target}^2}{2}\exp\left[-\frac{1}{2}\left(\frac{x_{target} - x_t}{0.71}\right)^2\right] \tag{5.19}$$

This function is null when far from the target position and unitary when the drone reaches it: the role of this second term is to penalize large velocities once the goal is approached by the drone.
The discounted reward-to-go to compute the advantage is defined as

$$\hat{R}_t = -\frac{1}{T}\sum_{k=t}^{T} \gamma^k r_k \quad \text{with} \quad [0, ..., t, ..., T] \tag{5.20}$$

(note that for $t = 0$, $\hat{R}_0 = R$ corresponds to the reward for the whole trajectory).

Now, considering a critic network V_ψ, the algorithm reads as shown in Algorithm 1.

Since this algorithm deploys a stochastic policy, the output of each RBF defines the mean of a Gaussian distribution with variance σ^2. Actions are thus sampled from the distribution

$$\boldsymbol{u}(\boldsymbol{x}) \sim \mathcal{N}(\boldsymbol{\pi}_\theta(\boldsymbol{x}), \sigma^2 I) \tag{5.21}$$

and together, the two RBFs form the Actor Network; similarly to the previous test case, they possess same number of centers and weights. The Critic Network is also implemented as an RBF network with parameters ψ, which takes in input the state and outputs the value function $V^\pi = V_\psi(\mathbf{x})$. All the parameters are trainable except the bias terms w_0, as depicted in Table 6.2.

Algorithm 1 Policy gradient algorithm

```
Input: initial policy parameters for policy $\theta_0$ and value function $\psi_0$
for k = 0, 1, 2 ... do
    Collect a set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running the policy $\pi_k = \pi(\theta_k)$ in the environment
    Compute reward-to-go $\hat{R}_t$ and policy ratio $\tilde{r}_\theta$
    Compute advantage estimate using the current value function $V_{\psi_k}$: $A_t = V_{\psi_k}(s_t) - \hat{R}_t$
    Compute surrogate losses $J_1^{surr}$ and $J_2^{surr}$
    Compute clipped objective
        $J^{CLIP}(\theta_k) = \mathbb{E}[\min(J_1^{surr}, J_2^{surr})]$
    Update policy parameter $\theta_k$ via Adam gradient ascent on $J^{CLIP}(\theta_k)$
    Fit value function parameter $\psi_k$ via regression on mean-squared error (gradient descent):
        $\psi_{k+1} \leftarrow \arg\min_\psi \frac{1}{T|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_{\psi_k}(s_t) - \hat{R}_t \right)^2$
end for
```

Chapter 6

Simulations and Results

6.1 ADEnv

The goal of the first part of this project is to compare the performance of the model based and model free controllers in the first simplified one dimensional dynamical system (ADEnv). This problem involves the control of the micro-drone assumed to be a dimensionless point mass, subject to lift/thrust, air drag and gravitational force.

6.1.1 Model Based simulation

The operating point $(\mathbf{x}_*, \mathbf{u}_*)$ is chosen to be $([2, 0]^T, [A_*, f_*]^T)$ where

$$A_* = 60^\circ \quad f_* = 40 \text{ Hz}. \tag{6.1}$$

and thus the controller tries to stabilize the error system $(\Delta\mathbf{x}, \Delta\mathbf{u}) = (\mathbf{x} - \mathbf{x}_*, \mathbf{u} - \mathbf{u}_*)$ at $(\mathbf{0}, \mathbf{0})$.
The matrices $\mathbf{A}$ and $\mathbf{B}$ come from equations (4.13) and (4.22), while $\mathbf{Q}$ and $\mathbf{R}$ are chosen according to the *diagonal weight* rule (4.9):

$$\mathbf{Q} = \begin{bmatrix} 10 & 0 \\ 0 & 0.1 \end{bmatrix} \qquad \mathbf{R} = 0.01 \begin{bmatrix} 10 & 0 \\ 0 & 5 \end{bmatrix} \tag{6.2}$$

Since we are considering an operating point, the LQR gain $\mathbf{K}$ is constant throughout the trajectory and is obtained by solving the algebraic Riccati equation with the set of matrices $(\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R})$.

Figure 6.1 shows the trajectory of the model based simulation in the error coordinate system ($\Delta\mathbf{x}_0 = [-2{,}0]^T$ and $\Delta\mathbf{x}_* = [0{,}0]^T$). As expected, the controller stabilizes the system around the operating point within approximately 1 s and then hovers at nominal conditions. The controls follow an optimal trend: they start at a higher values, decrease to a minimum, and then return to nominal conditions. The step-wise behavior of the controller is due to the fact that frequency and amplitude change only at the end of each cycle and therefore remain constant throughout it.

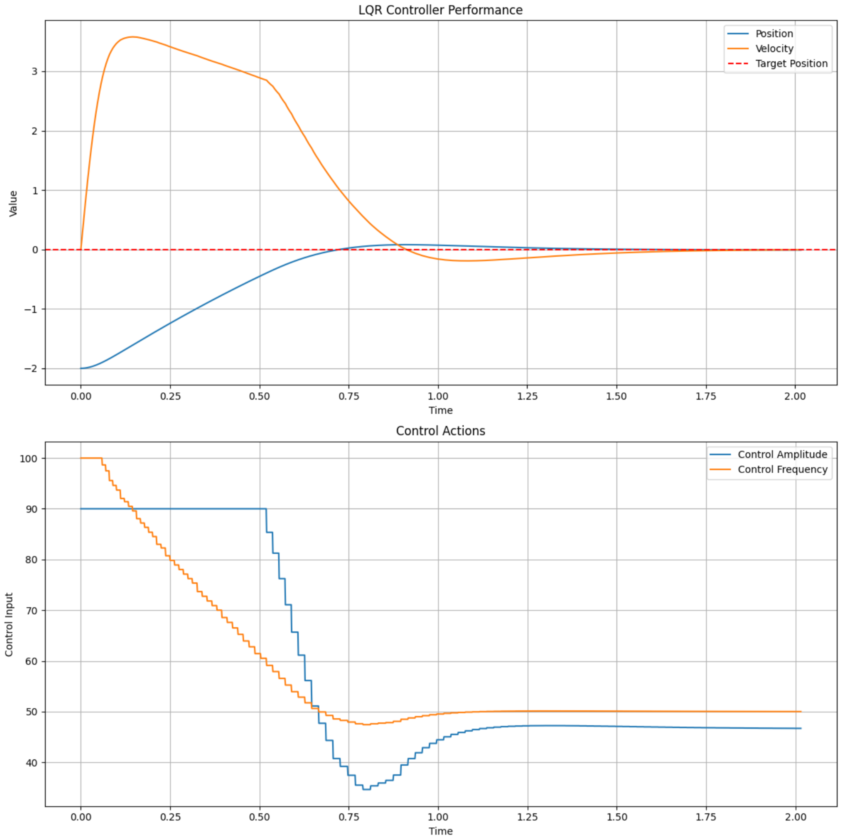


Figure 6.1: Model Based Trajectory

In Figure 6.2 the controller's performance can be further appreciated, by plotting each control input ($\Delta\mathbf{u} = \mathbf{u} - \mathbf{u}_*$) over state variable. The general trend is the same: the controller manage to stabilize the system at $(\Delta\mathbf{x}, \Delta\mathbf{u}) = (\mathbf{0}, \mathbf{0})$, successfully completing its task.

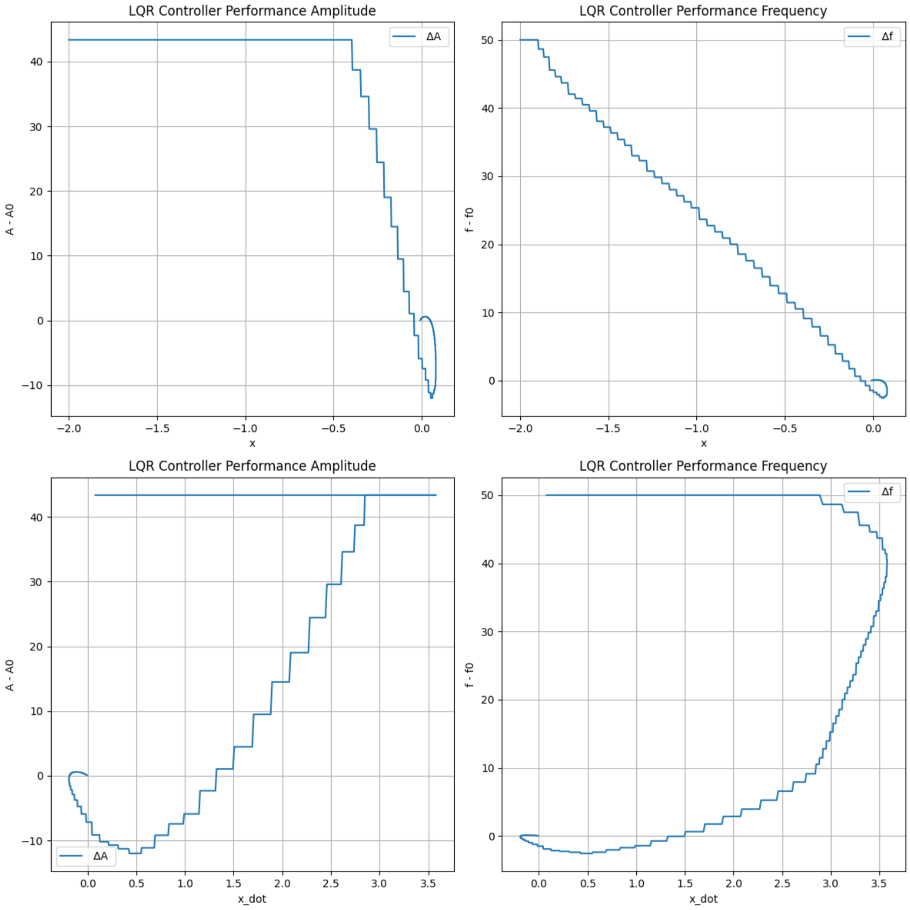


Figure 6.2: Control input $\Delta\mathbf{u}$ over state $\Delta\mathbf{x}$

6.1.2 Model Free simulation

The model free controller uses two RBFs to model the policies $[\pi_A, \pi_f]$, initialized as shown in Table 6.1 where Init. indicates the initial value and LR indicates the learning rate. Here, the two biases A_0 and f_0 are the same as $\mathbf{u}_* = [A_*, f_*]^T$ used in the model based simulation.

As mentioned earlier, in the ADEnv environment training involved directly optimizing the RBFs parameters using the Adam optimizer (with learning rates listed in Table 6.1), while varying the initial conditions (i.e. starting points $\mathbf{x}_0 = [x, 0]$) to encourage broader exploration and optimize over a wider range of positions. Since the simulations start from different

	π_A		π_f	
Parameter	Init.	LR	Init.	LR
centers along x	n = 100	$0.25 \cdot 10^{-4}$	n = 100	$0.25 \cdot 10^{-4}$
centers along $\dot{x}$	m = 100	$0.25 \cdot 10^{-4}$	m = 100	$0.25 \cdot 10^{-4}$
ϵ_x	2	10^{-5}	2	10^{-5}
$\epsilon_{\dot{x}}$	2	10^{-5}	2	10^{-5}
$\boldsymbol{w}$	0 deg	$0.5 \cdot 10^{-3}$	0 Hz	$0.5 \cdot 10^{-3}$
w_0	A_0	0	f_0	0

Table 6.1: ADEnv RBF Parameters

positions, presenting a standard learning curve would not be meaningful, as the rewards after each episode (which depend on the trajectory) vary significantly. The training process is therefore illustrated in Figure 6.3, where the final distance reached (on the y-axis) is plotted against the number of episodes. The first quarter of optimization shows the controller is still exploring and adapting, while after approximately 50 episodes, it begins to stabilize. Convergence to the target distance of 2 m at the end of each simulation is achieved after around 150 episodes.
After training, the RBF policies resemble those in Figure 6.4.

It's important to note that here the control inputs π_A and π_f are absolute values, unlike the $\Delta\mathbf{u}$ used in the MB simulation. The biases A_0 and f_0 are used only to initialize the optimization: to compute $\bar{F}_p$ as in equation (4.14), we use directly $\mathbf{u}(\mathbf{x}) = [\pi_A(\mathbf{x}), \pi_f(\mathbf{x})]$.
The model free trajectory in Figure 6.5 (in the error coordinate system $\Delta\mathbf{x}_0 = [-2{,}0]^T$ and $\Delta\mathbf{x}_* = [0{,}0]^T$) shows the drone reaching its target and hovering, while Figure 6.6 displays the control variables against states.

Similarly to the model based case, the controller manages to stabilize the system to the desired point.

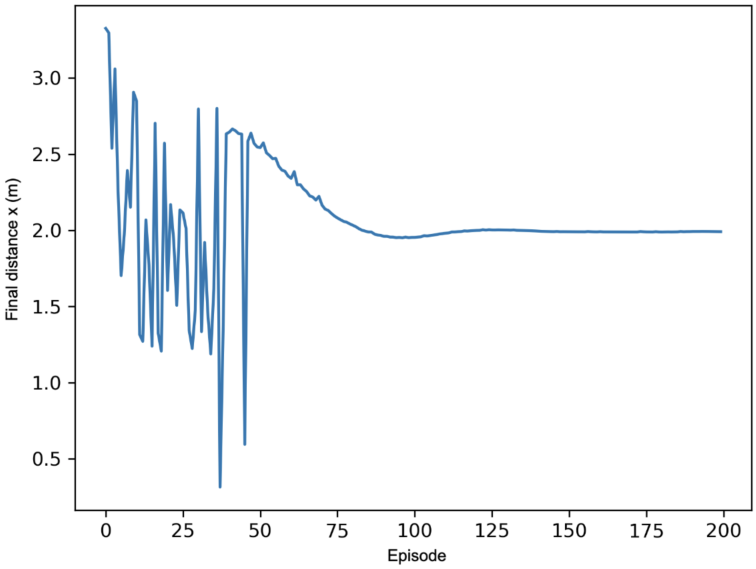


Figure 6.3: Final distance over episode

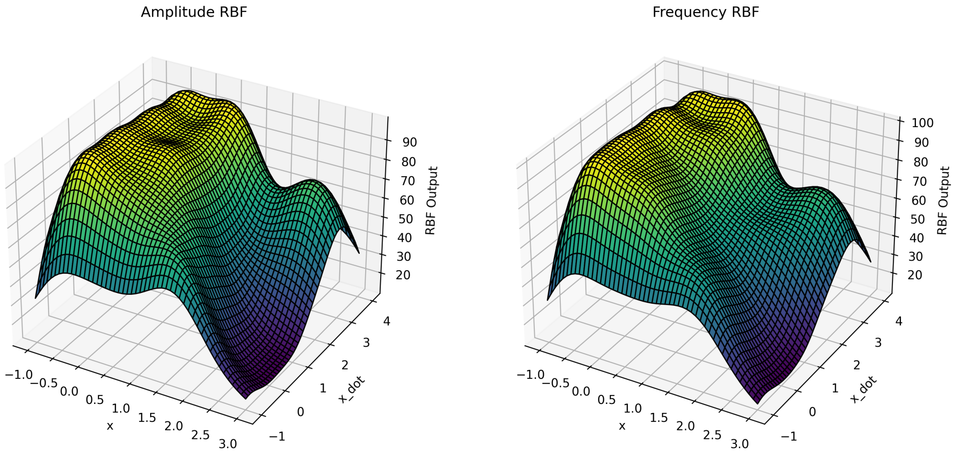


Figure 6.4: ADEnv: trained RBF policies

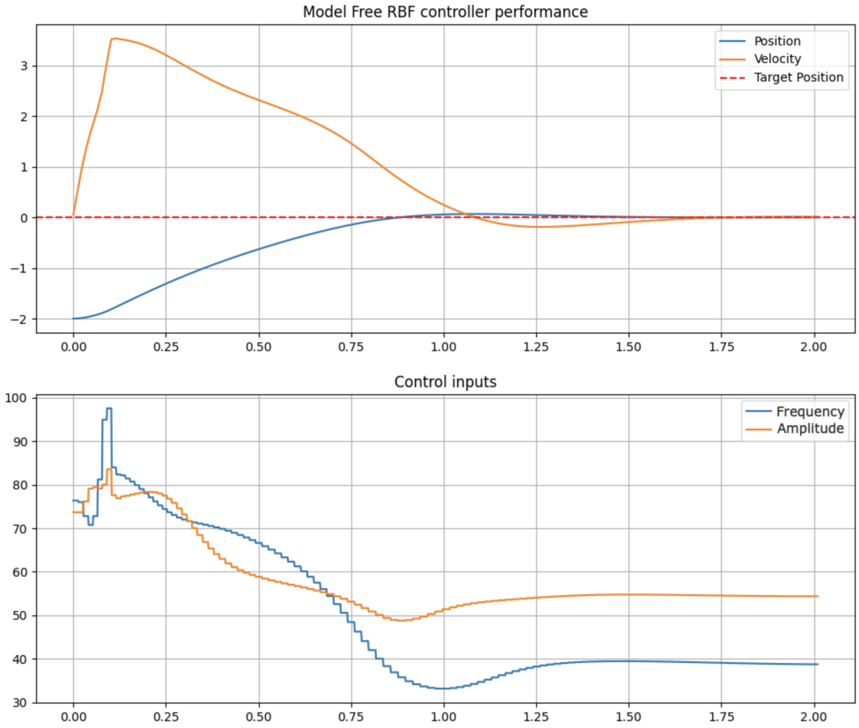
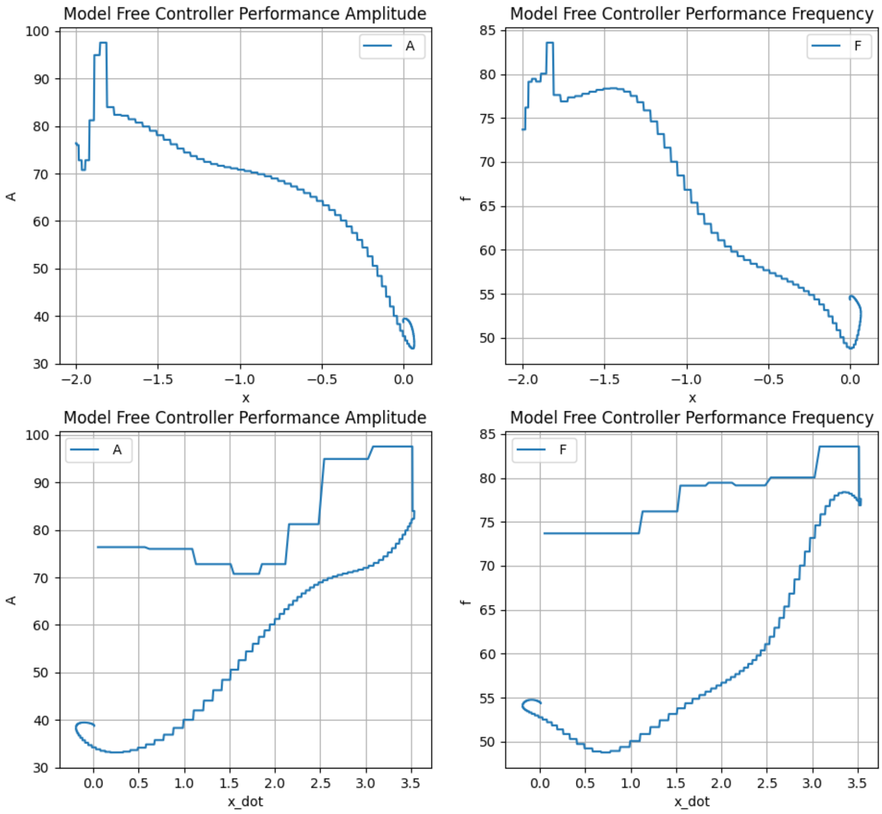


Figure 6.5: Model Free Trajectory

6.1.3 Comparison

Both controllers produce similar trajectories, reaching the target in approximately 1 s and stabilizing the system in similar times. The model free controller optimized a policy that approximates the model based optimal one, though small differences exist:

- The model free controller shows more variation in the control inputs, especially in the early phase.
- The hovering conditions differs: considering the equation of $\bar{F}_p$ (4.14), infinitely many pairs of $[A, f]$ can produce the thrust needed to hover (i.e. balancing drag and gravitational force), and during training one such pair is randomly selected. Furthermore, while the LQR cost


Model Free Controller Performance Amplitude

Model Free Controller Performance Frequency

Model Free Controller Performance Amplitude

Model Free Controller Performance Frequency

Figure 6.6: Control input π over state x

function (4.2) penalizes both states and control inputs, the model-free reward function (5.17) considers only the state variables.

Noise robustness was also tested by introducing white (Gaussian) and correlated (OU) noise. The noisy input signal was generated by evaluating $\mathbf{u}(\mathbf{x} + \mathbf{w})$ to produce the control, where $\mathbf{w}$ is the noise, as explained in Section 3.1.1. In the discrete-time setting, the noise is defined as:

- **White**: $w_t = \sigma_0 \eta$

- **Correlated**: $w_{t+1} = -\theta w_t + \sigma_0 \eta$

All simulations were conducted by increasing the parameter σ_0, with 20 episodes executed for each value of σ_0, and evaluating the mean reward at the end to assess performance.
The reward function used in this test is defined as

$$R = \sum_{t=0}^{T} \|\Delta x\|_2^2 + 0.01\|\Delta \dot{x}\|_2^2 \tag{6.3}$$

where a lower reward indicates closer adherence of the controller to the reference trajectory.
Figure 6.7 illustrates the results with white noise, showing that the model based controller exhibits higher robustness and lower initial cost compared to the model free.

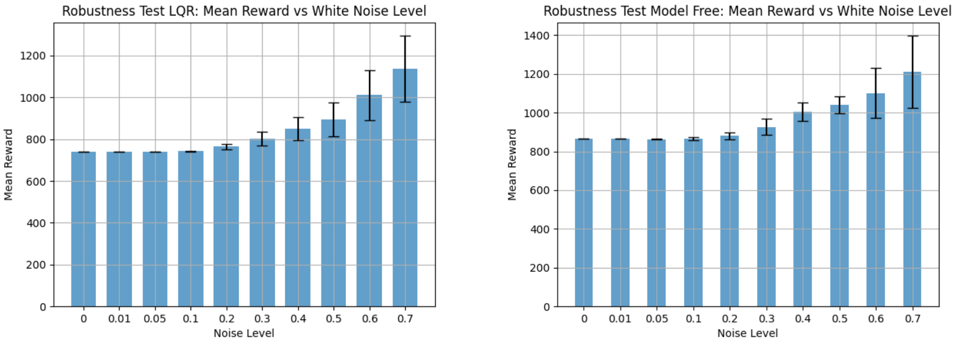


Figure 6.7: White noise robustness

Figure 6.8 shows the controllers' behavior with correlated noise: again, the trend is similar and the model based controller outperforms the model free one.

6.2 FDEnv

The goal of this section is to evaluate whether the RBF policy can learn to control the more complex environment using non-state-of-the-art algorithms such as REINFORCE and its enhanced version (Algorithm 1).

The RBF parameters are listed in Table 6.2. Using the normalizing function defined in (3.15), the bias terms A_0 and f_0 correspond to a

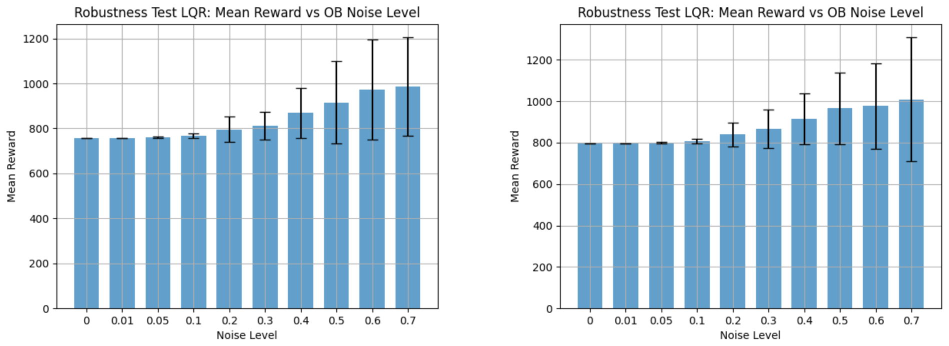


Figure 6.8: Correlated noise robustness

	π_A		π_f		V^π	
Parameter	Init.	LR	Init.	LR	Init.	LR
centers along x	n = 200	$0.25 \cdot 10^{-5}$	n = 200	$0.25 \cdot 10^{-5}$	n = 200	$0.25 \cdot 10^{-5}$
centers along $\dot{x}$	m = 200	$0.25 \cdot 10^{-5}$	m = 200	$0.25 \cdot 10^{-5}$	m = 200	$0.25 \cdot 10^{-5}$
ϵ_x	1.43	10^{-5}	1.43	10^{-5}	1.43	10^{-4}
$\epsilon_{\dot{x}}$	1.43	10^{-5}	1.43	10^{-5}	1.43	10^{-4}
$\boldsymbol{w}$	0	$0.25 \cdot 10^{-4}$	0	$0.25 \cdot 10^{-4}$	0	10^{-4}
w_0	A_0	0	f_0	0	0	0

Table 6.2: FDEnv RBF Parameters

normalized amplitude of 50° and normalized frequency of 30 Hz:

$$A_0 = \text{normalize}(50, A_{\phi,min}, A_{\phi,max})^\circ$$
$$f_0 = \text{normalize}(30, f_{min}, f_{,max})\text{Hz}.$$

As before, the learnable parameters are the shape factors $\boldsymbol{\epsilon}$, the centers $\mathbf{x}_i$ and the weights $\boldsymbol{w}$. Due to the complexity of this environment, training with varying initial positions resulted in poorer performance and slower convergence: this is likely due to the on-policy nature and naivety of the algorithm, which struggled to discover effective action sequences in a consistent way, leading to high variance in the returns. Thus, training was conducted with a fixed initial position. The goal is to reach the target point $\mathbf{x}_{target} = [5, 0]$ starting from $\mathbf{x}_0 = [0, 0]$, which after normalization and in the error coordinate system read $\mathbf{e}_{target} = [0, 0]$ and $\mathbf{e}_0 = [-1, 0]$.

6.2.1 Training and Results

The first algorithm tested was Vanilla Policy Gradient. Despite the simplicity of the RBF approximator, the agent did not succeed to learn due to the high variance of policy gradient updates: Figure 6.9 shows how even after 1000 episodes, training did not even start to stabilize. This behavior is likely to the fact that the rewards were always negative, causing the probability of each action to be consistently reduced (see Section 5.1.2).

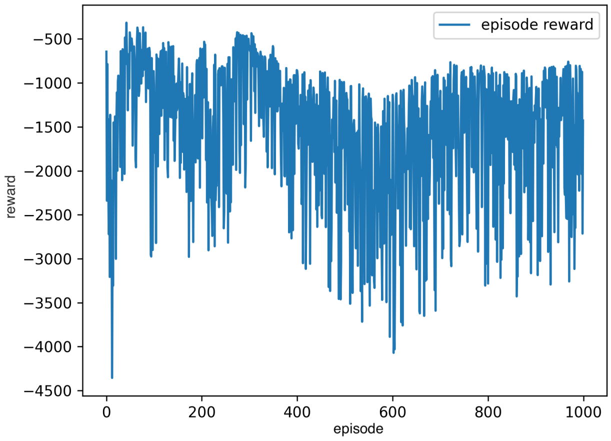


Figure 6.9: REINFORCE unsuccessful training

Therefore, Algorithm 1 was employed. Figure 6.10 shows the learning curve over 2500 episodes. It can be observed that after approximately 1500 episodes convergence occurs, with limited performance improvements in the remaining iterations.

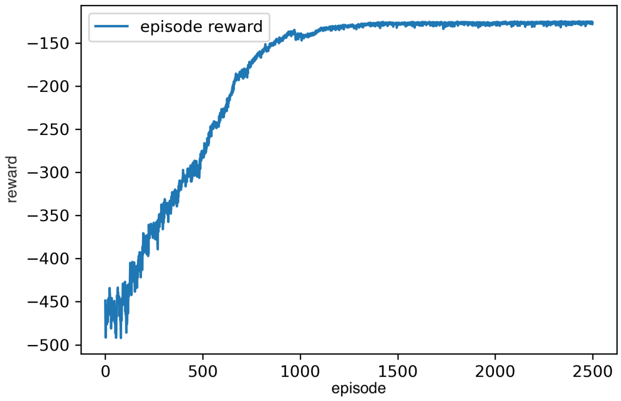


Figure 6.10: Learning curve - 2500 episodes

Although training was relatively slow and sample inefficient, the goal was not efficient training but rather to test the effectiveness of RBF-based policy representation. Several factors contributed to the long training time:

- The algorithm used is relatively naive, as it does not consider entropy regularization, experience buffer replay, or batch learning.
- The learning process was on-policy, making it intrinsically sample inefficient.
- Hyper-parameter tuning was not performed: given the simplicity of RBFs, a trial-and-error approach was sufficient to obtain satisfactory results. However, it is well known that implementation details and parameters such as learning rates significantly impact training stability and convergence speed (see Engstrom et al. [36]).
- In the ADEnv, due to the averaging effect, the influence of the flapping frequency appears in the system's equation only through the control input. In contrast, the FDEnv explicitly models the instantaneous wing forces (directly dependent on flapping frequency) which continuously influence the drone's trajectory during each cycle. As a result, changes in frequency have a much stronger and direct impact on the FDEnv's dynamics and influence more the learning process.

The evolution of the policy during training, together with the corresponding trajectory, is presented in Section 6.2.2, where Figure 6.18 shows the final trained policy. A trajectory from a 2.5 s simulation is presented in Figure 6.11, demonstrating successful target reaching and stable hovering at almost null velocity. As expected, the control inputs exhibit a behavior similar to that observed in the ADEnv environment (Figure 6.5): they start at higher values, gradually decrease and eventually stabilize during hovering. Figure 6.12 shows the Action-State relationship where, again, the controller manage to stabilize the drone near the target position.

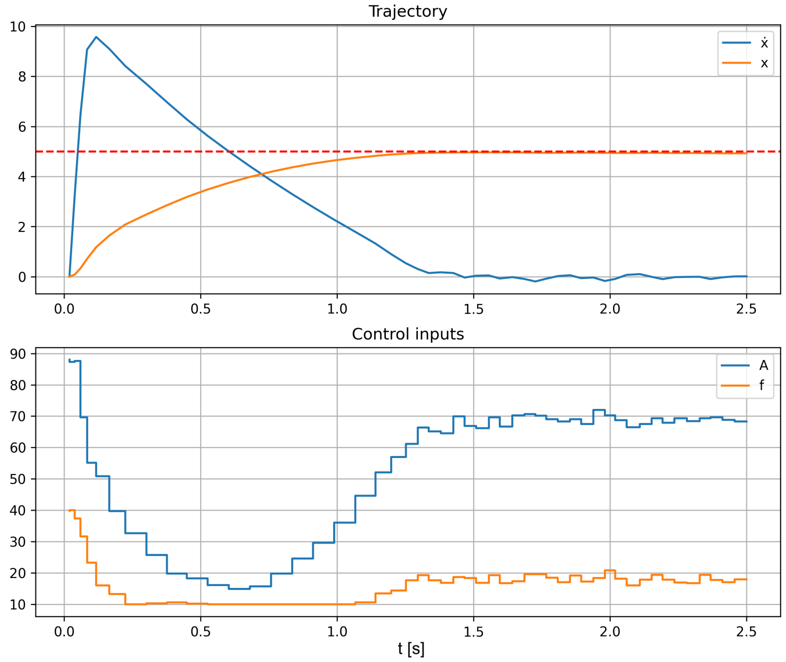


Figure 6.11: Trajectory after training

6.2.2 RBF updates

This section illustrates the evolution of RBF policies across different episodes within a single training session. Beneath the RBFs, the curve shows the drone's phase-space trajectory in the error coordinate system

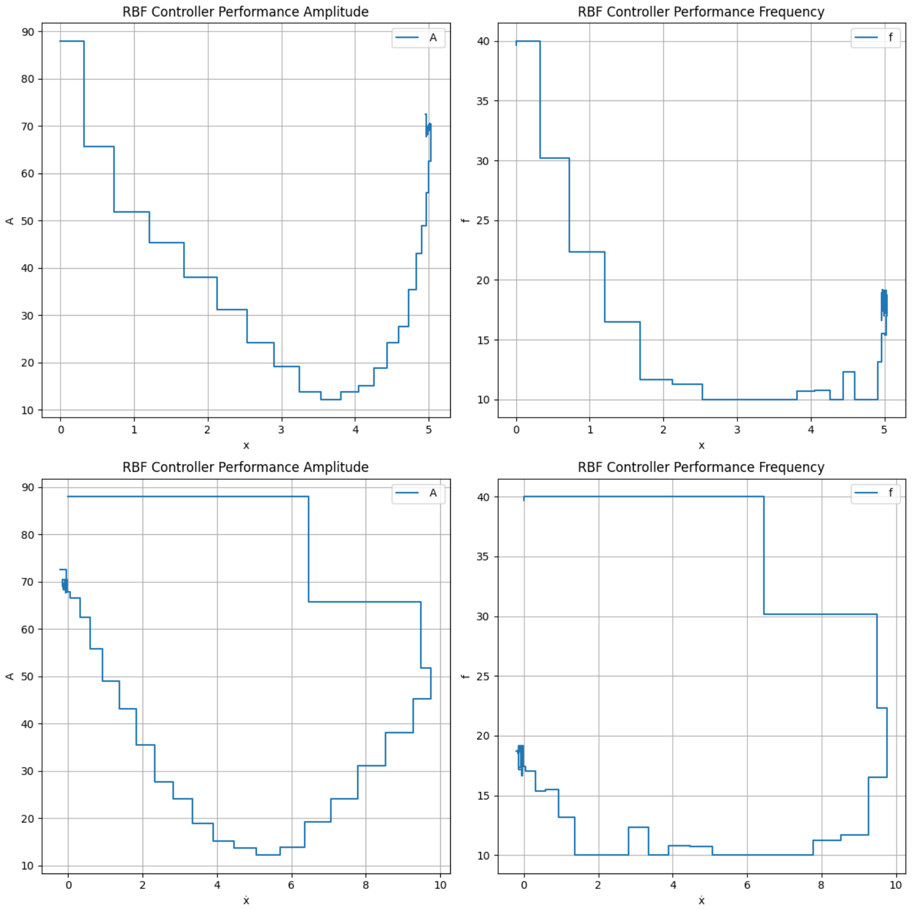


Figure 6.12: Control input π over state x

($\mathbf{e}_0 = [-1, 0]^T$ and $\mathbf{e}_{\text{target}} = [0, 0]^T$); black dots indicate the end of each flapping cycle, when a new action is generated by the policy π_θ. It can be appreciated how similar are Figures 6.17 and 6.18, especially in terms of trajectory.

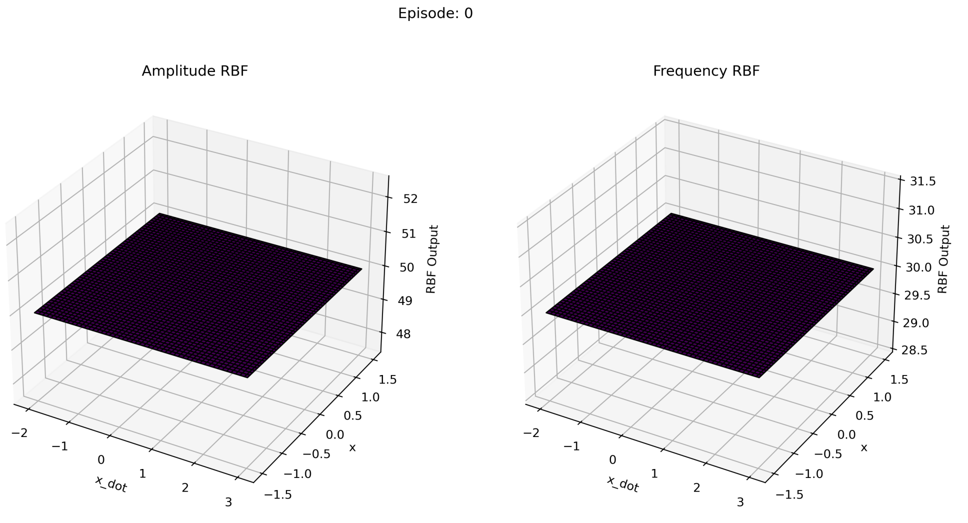


Figure 6.13: RBFs initialization

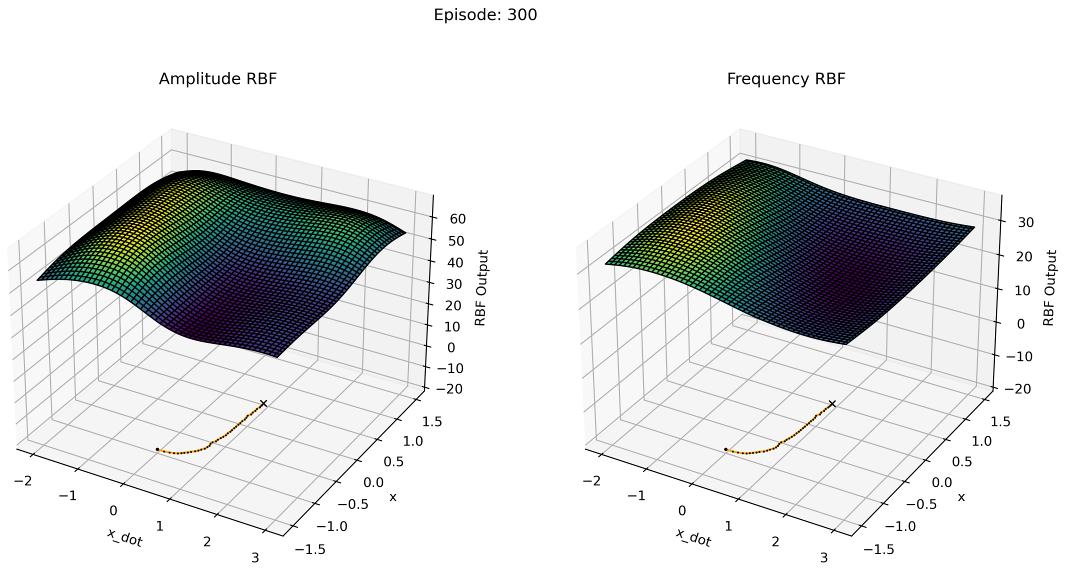


Figure 6.14: RBFs after 300 episodes

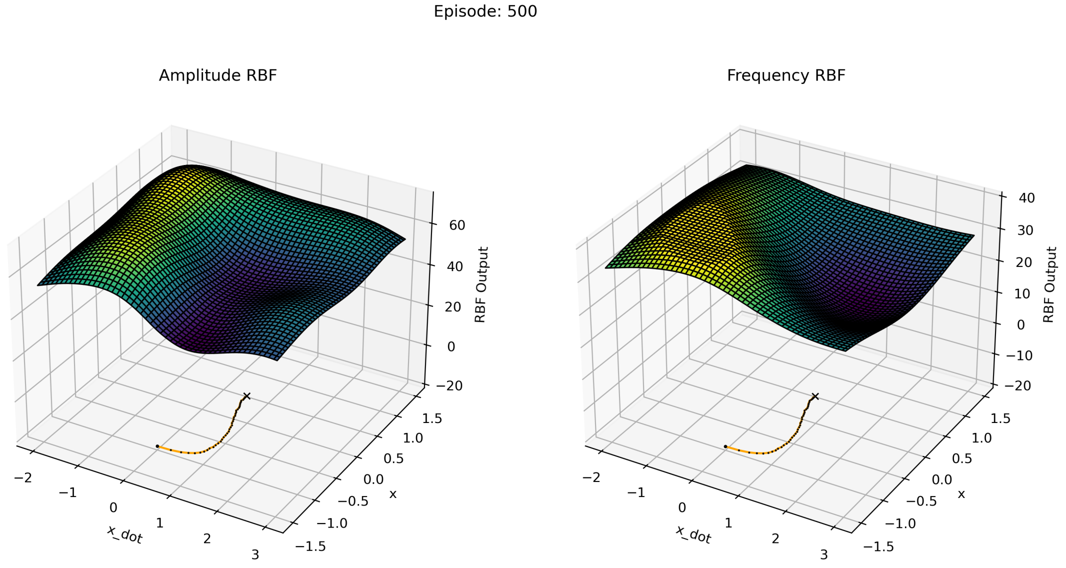


Figure 6.15: RBFs after 500 episodes

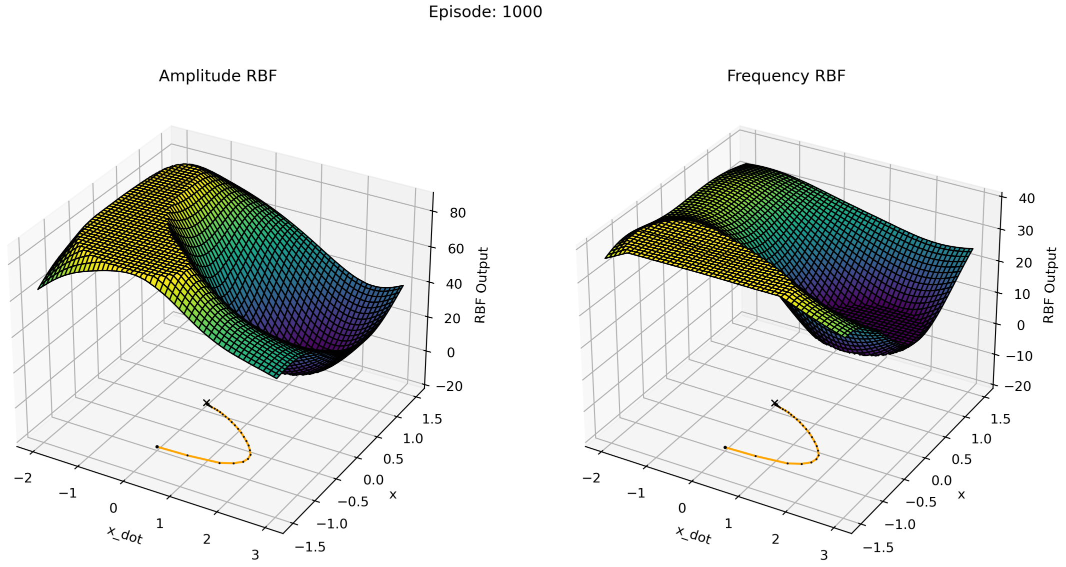


Figure 6.16: RBFs after 1000 episodes

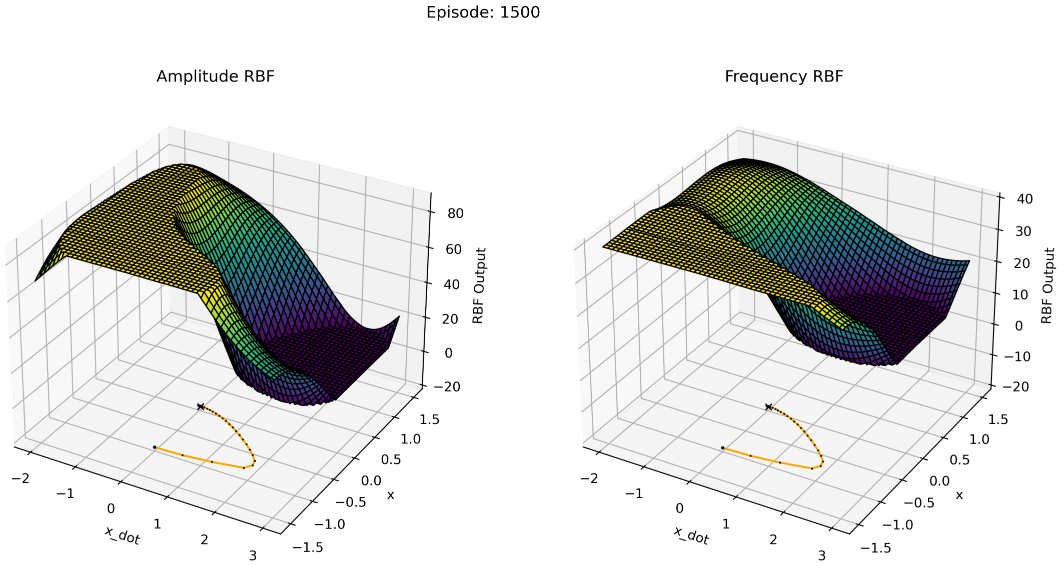


Figure 6.17: RBFs after 1500 episodes

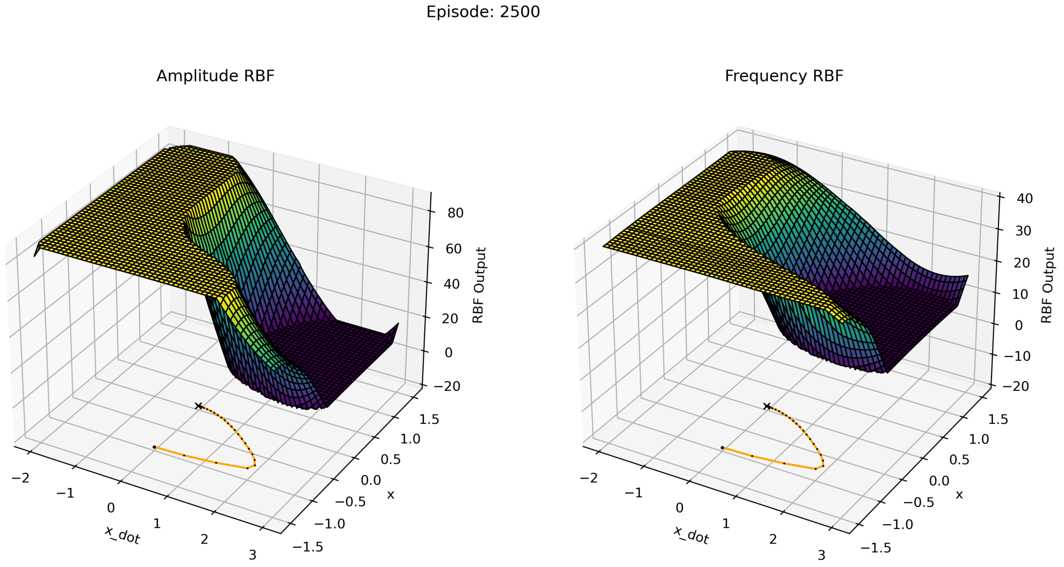


Figure 6.18: RBFs after 2500 episodes

Robustness

Robustness was tested using the same method as in the first test case: noise disturbances were introduced, and the agent's performance was assessed under these conditions. Results are depicted in Figure 6.19:

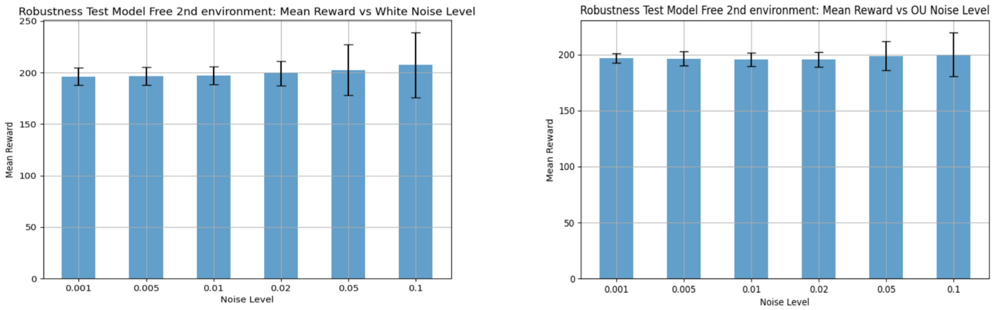


Figure 6.19: Second test case robustness: white noise (on the left) and OU noise (on the right)

While increasing the σ_0 parameter, the controller manages to obtain satisfactory result, without significantly compromising the performances. To further test generalization capabilities, considering that training was performed starting from the same point, the trained policy was also evaluated from different initial positions $\mathbf{x}_0 = [x_0, \dot{x}_0]^T$:

$$\mathbf{x}_0 = \left\{ \begin{bmatrix} -1.5 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.7 \\ 0 \end{bmatrix}, \begin{bmatrix} 2.2 \\ 0 \end{bmatrix}, \begin{bmatrix} 3.7 \\ 0 \end{bmatrix}, \begin{bmatrix} 6.3 \\ 0 \end{bmatrix}, \begin{bmatrix} 7.1 \\ 0 \end{bmatrix} \right\} \tag{6.4}$$

Figure 6.20 presents the corresponding trajectories in the nominal coordinate system, where the controller successfully reaches to the target in all cases. Simulations lasting $8s$ were required for initial states above the target to reach it, due to constraints on flapping amplitude ($A_{\phi,min} = 10°$) which prevented immediate free fall and thus required more time.

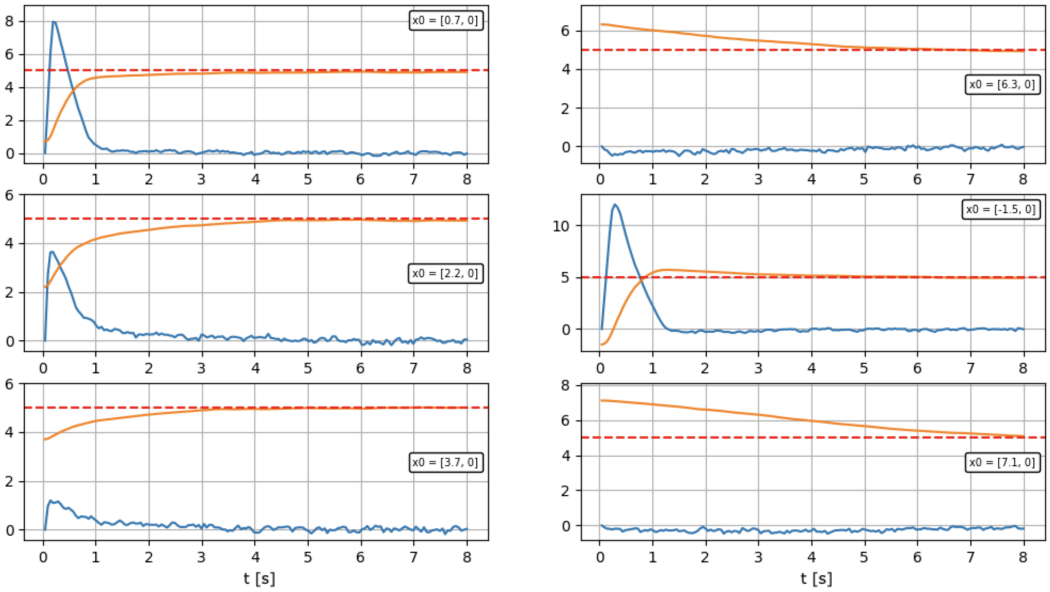


Figure 6.20: Trajectories with different initial points

Chapter 7

Conclusions

This thesis explored the applicability of model free controller, whose policy is modeled using Radial Basis Functions, to the one dimensional motion of Flapping-Wing Micro Air Vehicles. First, a cycle averaged model was developed, allowing the model free optimization of the RBF and its comparison with traditional LQR controller. Then, with a more complex model, the RBF policy was successfully trained using basic reinforcement learning algorithms.

The simulations demonstrate that RBF-based policies can effectively learn control strategies for complex environments, even when directly optimized or trained using basic policy gradient methods.
Hyper-parameter selection was relatively straightforward: given the simplicity of the function approximator and knowing its physical meaning, selecting bias terms and weight magnitudes was intuitive.

In the averaged dynamics test case, the RBF controller successfully optimized nearly-optimal policy after only 150 episodes, achieving similar performances to the model based controller.

In the more challenging non-averaged (full dynamics) test case, the policy gradient algorithm converged after 1500 episodes, where a model based approach such as LQR would have been troublesome to implement, considering the non-linearities and modeling problems.

The controllers achieved stable hovering behavior and demonstrated robustness against both noise and varying initial conditions. However, performance is limited due to the inefficiency of the learning algorithm, and the controller is not yet suited for real world application. Future improvements may include:

- Entropy regularization, to reduce variance
- Adaptive learning rates, to achieve faster convergence
- Hyper parameter tuning
- State-of-art, sample efficient reinforcement learning algorithms integration.

Nonetheless, these result confirm that RBFs can serve as a viable alternative to neural networks for reinforcement learning in low-dimensional control tasks.

Appendix A

Appendices

A.1 Policy Gradient Theorem

The policy gradient theorem gives a fundamental tool in reinforcement learning, describing how the expected return's gradient depends on the policy parameters.

Considering a stochastic policy $\pi_\theta(a \mid s)$ parameterized by θ, with states s and actions a, the objective function is the expected return:

$$J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}[r(\tau)] = \int_\tau p_\theta(\tau) R(\tau) d\tau., \tag{A.1}$$

where $p_\theta(\tau)$ is the probability distribution over trajectories induced by the policy and environment dynamics[1].

Differentiating $J(\theta)$:

$$\nabla_\theta J(\theta) = \nabla_\theta \int_\tau p_\theta(\tau) R(\tau) d\tau. \tag{A.2}$$

Using the identity

$$\nabla log(x) = \frac{\nabla x}{x} \tag{A.3}$$

we get

[1] thus the probability of a trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ under the policy π_θ

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) \tag{A.4}$$

and, by plugging it back into (A.2) and exchanging integration and derivation

$$\nabla_\theta J(\theta) = \int_\tau p_\theta(\tau) R(\tau) \nabla_\theta \log p_\theta(\tau) d\tau. \tag{A.5}$$

Rewriting as expectation:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[R(\tau)\nabla_\theta \log p_\theta(\tau)\right]. \tag{A.6}$$

we now expand $p_\theta(\tau)$ in terms of state-action transitions

$$p_\theta(\tau) = p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t), \tag{A.7}$$

we get[2]

$$\log p_\theta(\tau) = \log p(s_0) + \sum_{t=0}^{T-1} \log p(s_{t+1}|s_t, a_t) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) \tag{A.8}$$

and, differentiating with respect to θ:

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) + \sum_{t=0}^{T-1} \nabla_\theta \log p(s_{t+1}|s_t, a_t). \tag{A.9}$$

Now, considering that the environment dynamics does not depend on θ (thus also $p(s_{t+1}|s_t, a_t)$):

$$\nabla_\theta \log p(s_{t+1}|s_t, a_t) = 0 \tag{A.10}$$

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t). \tag{A.11}$$

Substituting (A.11) into (A.6) leads to the standard policy gradient theorem:

[2] using $log(ab) = log(a) + log(b)$

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]. \tag{A.12}$$

Having an expression for the gradient, it's now possible to leverage on standard gradient based optimization techniques (such as gradient ascent for REINFORCE) to optimize policy parameter. It's important to note also that the theorem (such as the algorithm) is agnostic to the function approximator used to model $\pi_\theta(a \mid s)$, which makes it a very powerful tool.

A.2 Introducing a baseline b

In order to reduce variance in policy gradient estimates, it is possible to introduce a baseline function $b = b(s)$; it is important that this baseline depends only on the states and not on the actions. To prove that introducing b does not change the results of the policy gradient theorem, we start from (A.12) and we substitute $R(\tau)$ with $R(\tau) - b(s)$ obtaining the modified expectation

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(R(\tau) - b)\right]. \tag{A.13}$$

By being the expectation operator linear, it is possible to rewrite it as:

$$\nabla J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right] - \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) b\right] \tag{A.14}$$

where first term is the original policy gradient. Focusing on the second term, considering that b does not depend on action a_t, it's possible to factor it out:

$$\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} b\nabla_\theta \log \pi_\theta(a_t|s_t)\right] = b\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\right]. \tag{A.15}$$

Now, remembering (A.11) we get that the second term reads

$$b \underset{\tau\sim\pi_\theta}{\mathbb{E}} \left[\nabla_\theta \log p_\theta(\tau)\right]. \tag{A.16}$$

By definition of expectation

$$b \int_\tau p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) d\tau \tag{A.17}$$

and, remembering the Log-derivative rule in (A.4) we obtain

$$b \int_\tau \nabla_\theta p_\theta(\tau) d\tau = b \nabla_\theta \int_\tau p_\theta(\tau) d\tau.^3 \tag{A.18}$$

By definition of probability distribution, $\int_\tau p d\tau = 1$ therefore

$$b \nabla_\theta \int_\tau p_\theta(\tau) d\tau = b \nabla_\theta 1 = 0 \tag{A.19}$$

and the second term vanishes

$$\underset{\tau\sim\pi_\theta}{\mathbb{E}} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) b\right] = 0, \tag{A.20}$$

obtaining the same result as (A.12). Thus, introducing a baseline does not change the expectation of the policy gradient, while helping to reduce its variance.

[3] exchanging integration over τ and derivation over θ

Bibliography

[1] Katsuhiko Ogata. *Modern Control Engineering*. 5th ed. Prentice Hall, 2010 (cit. on p. 1).

[2] Donald E. Kirk. *Optimal Control Theory: An Introduction*. Dover Publications, 1970 (cit. on p. 1).

[3] Zhao Jiaxin, Zhang Weiping, Wang Chenyang, Zou Yang, and Wang Jiahao. «Current Status of insect-inspired Flapping Wing Micro Air Vehicles». In: *2019 IEEE International Conference on Unmanned Systems (ICUS)*. 2019, pp. 894–898. DOI: 10.1109/ICUS48101.2019.8996060 (cit. on p. 2).

[4] Matthew Keennon, Karl Klingebiel, and Henry Won. «Development of the Nano Hummingbird: A Tailless Flapping Wing Micro Air Vehicle». In: *Proceedings of 50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. Jan. 2012. ISBN: 978-1-60086-936-5. DOI: 10.2514/6.2012-588 (cit. on pp. 2, 3).

[5] Mukhtar Hamza. «Introduction to Root Locus Method». In: (Feb. 2018). DOI: 10.13140/RG.2.2.14886.01602 (cit. on p. 7).

[6] Haowen Jiang. «Overview and development of PID control». In: *Applied and Computational Engineering* 66 (May 2024), pp. 187–191. DOI: 10.54254/2755-2721/66/20240946 (cit. on p. 7).

[7] Sebastian F. Tudor, Cristian Oara, and Serban Sabau. *H-infinity control problem for general discrete-time systems*. 2014. arXiv: 1403.6225 [math.OC]. URL: https://arxiv.org/abs/1403.6225 (cit. on p. 7).

[8] Andrzej Bartoszewicz and Justyna Zuk. «Sliding mode control — Basic concepts and current trends». In: *2010 IEEE International Symposium on Industrial Electronics*. 2010, pp. 3772–3777. DOI: 10.1109/ISIE.2010.5637990 (cit. on p. 8).

[9] Richard E. Kopp. «Pontryagin Maximum Principle». In: *Optimization Techniques*. Ed. by George Leitmann. Vol. 5. Mathematics in Science and Engineering. Elsevier, 1962, pp. 255–279. DOI: 10.1016/S0076-5392(08)62095-0. URL: https://www.sciencedirect.com/science/article/pii/S0076539208620950 (cit. on p. 8).

[10] Yunong Zhang. «A survey of dynamic programming algorithms». In: *Applied and Computational Engineering* 35 (Jan. 2024), pp. 183–189. DOI: 10.54254/2755-2721/35/20230392 (cit. on p. 9).

[11] David Silver et al. «Mastering the game of Go with deep neural networks and tree search». In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961 (cit. on p. 12).

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602 (cit. on p. 15).

[13] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG]. URL: https://arxiv.org/abs/1509.02971 (cit. on p. 16).

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html (cit. on p. 16).

[15] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: 1602.01783 [cs.LG]. URL: https://arxiv.org/abs/1602.01783 (cit. on p. 16).

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms.* 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347 (cit. on pp. 16, 42, 43).

[17] E. Wilson and D.W. Tufts. «Multilayer perceptron design algorithm». In: *Proceedings of IEEE Workshop on Neural Networks for Signal Processing.* 1994, pp. 61–68. DOI: 10.1109/NNSP.1994.366063 (cit. on p. 17).

[18] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735 (cit. on p. 18).

[19] Mohammadreza Amirian and Friedhelm Schwenker. «Radial Basis Function Networks for Convolutional Neural Networks to Learn Similarity Distance Metric and Improve Interpretability». In: *IEEE Access* 8 (2020), pp. 123087–123097. ISSN: 2169-3536. DOI: 10.1109/access.2020.3007337. URL: http://dx.doi.org/10.1109/ACCESS.2020.3007337 (cit. on p. 19).

[20] Mark JL Orr et al. *Introduction to radial basis function networks.* 1996 (cit. on p. 19).

[21] Ch Dash, Ajit Kumar Behera, Satchidananda Dehuri, and Sung-Bae Cho. «Radial basis function neural networks: A topical state-of-the-art survey». In: *Open Computer Science* 6 (May 2016). DOI: 10.1515/comp-2016-0005 (cit. on p. 19).

[22] Mark Ebden. *Gaussian Processes: A Quick Introduction.* 2015. arXiv: 1505.02965 [math.ST]. URL: https://arxiv.org/abs/1505.02965 (cit. on p. 19).

[23] Haithem Taha, Muhammad R. Hajj, and Ali Nayfeh. «Flight dynamics and control of flapping-wing MAVs: A review». In: *Nonlinear Dynamics* 70 (Oct. 2012). DOI: 10.1007/s11071-012-0529-5 (cit. on p. 20).

[24] Romain Poletti, Lilla Koloszar, and Miguel Alfonso Mendez. «On the Flight Control of Flapping Wing Micro Air Vehicles with Model-Based Reinforcement Learning». In: *Proceedings* 107.1 (2024). ISSN:

2504-3900. DOI: 10.3390/proceedings2024107033. URL: https://www.mdpi.com/2504-3900/107/1/33 (cit. on pp. 24, 25, 28, 29).

[25] John Whitney and R. Wood. «Aeromechanics of passive rotation in flapping flight». In: *Journal of Fluid Mechanics - J FLUID MECH* 660 (Oct. 2010), pp. 197–220. DOI: 10.1017/S002211201000265X (cit. on p. 24).

[26] Y Lee, Kim Boon Lua, T Lim, and K. Yeo. «A quasi-steady aerodynamic model for flapping flight with improved adaptability». In: *Bioinspiration and Biomimetics* 11 (Apr. 2016), p. 036005. DOI: 10.1088/1748-3190/11/3/036005 (cit. on pp. 27, 29).

[27] Sanjay Sane. «The aerodynamics of insect flight». In: *The Journal of experimental biology* 206 (Jan. 2004), pp. 4191–208. DOI: 10.1242/jeb.00663 (cit. on p. 29).

[28] Rudolf E. Kalman. «Contributions to the theory of optimal control». In: *Boletin de la Sociedad Matematica Mexicana* 5.2 (1960), pp. 102–119 (cit. on p. 31).

[29] Riccardo Poli, James Kennedy, and Tim Blackwell. «Particle Swarm Optimization: An Overview». In: *Swarm Intelligence* 1 (Oct. 2007). DOI: 10.1007/s11721-007-0002-0 (cit. on p. 33).

[30] Peter I. Frazier. *A Tutorial on Bayesian Optimization.* 2018. arXiv: 1807.02811 [stat.ML]. URL: https://arxiv.org/abs/1807.02811 (cit. on p. 33).

[31] Ronald J. Williams. «Simple statistical gradient-following algorithms for connectionist reinforcement learning». In: *Machine Learning* 8.3 (1992), pp. 229–256. ISSN: 1573-0565. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696 (cit. on p. 38).

[32] Hsiao-Ru Pan, Nico Gürtler, Alexander Neitz, and Bernhard Scholkopf. *Direct Advantage Estimation.* 2023. arXiv: 2109.06093 [cs.LG]. URL: https://arxiv.org/abs/2109.06093 (cit. on p. 41).

[33] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. *High-Dimensional Continuous Control Using Generalized Advantage Estimation.* 2018. arXiv: 1506.02438 [cs.LG]. URL: https://arxiv.org/abs/1506.02438 (cit. on p. 41).

[34] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. *Trust Region Policy Optimization*. 2017. arXiv: `1502.05477 [cs.LG]`. URL: `https://arxiv.org/abs/1502.05477` (cit. on p. 42).

[35] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`. URL: `https://arxiv.org/abs/1412.6980` (cit. on p. 47).

[36] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. *Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO*. 2020. arXiv: `2005.12729 [cs.LG]`. URL: `https://arxiv.org/abs/2005.12729` (cit. on p. 60).

[37] Christopher JCH Watkins and Peter Dayan. «Q-learning». In: *Machine learning* 8 (1992), pp. 279–292.

[38] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. arXiv: `1801.01290 [cs.LG]`. URL: `https://arxiv.org/abs/1801.01290`.

[39] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. *Neural Ordinary Differential Equations*. 2019. arXiv: `1806.07366 [cs.LG]`. URL: `https://arxiv.org/abs/1806.07366`.

[40] Haider N, Shahzad A, Mumtaz Qadri MN, and Ali Shah SI. «Recent progress in flapping wings for micro aerial vehicle applications». In: *Proceedings of the Institution of Mechanical Engineers, Part C* 235.2 (2020), pp. 245–264. DOI: `10.1177/0954406220917426`.

[41] Erica Kim, Marta Wolf, Victor Ortega-Jimenez, Stanley Cheng, and Robert Dudley. «Hovering performance of Anna's Hummingbirds (Calypte anna) in ground effect». In: *Journal of the Royal Society, Interface / the Royal Society* 11 (Sept. 2014). DOI: `10.1098/rsif.2014.0505`.