Sparse Hardware-Efficient Control of the DelFly

Bachelor Thesis

P.A. Bakker & A.H. Mohammad

Delft University of Technology

Towards Sparse Hardware-Efficient Control of the DelFly

Control Algorithm

by

P.A. Bakker and A.H. Mohammad

Student name	Student number
Pablo A. Bakker	5815193
Amir H. Mohammad	5839475

Supervisor: Dr. C. Frenkel, TU Delft

Thesis Committee: Dr. C. Frenkel, TU Delft
Dr. A. Yarovoy, TU Delft
Dr. M.A Del Pino, TU Delft

Delft University of Technology
Faculty of EEMCS – BSc Electrical Engineering
Thesis Defence: 25 June 2025

Abstract

Flapping-wing micro air vehicles (FWMAVs) present a significant control challenge due to their complex nonlinear dynamics and severe hardware constraints, which preclude the use of computationally intensive controllers. This thesis addresses this challenge by developing and validating a pipeline to convert a high-performance neural network policy, trained via Reinforcement Learning (RL), into a sparse, hardware-efficient symbolic controller using the Sparse Identification of Nonlinear Dynamics (SINDy) framework.

The primary contribution of this work is the introduction and evaluation of novel, hardware-aware optimizations within the SINDy distillation process. Specifically, we introduce Sparse Bit Quantization (SBQ), a new quantization scheme that represents coefficients as combinations of powers of two to enable efficient implementation using bit-shift operations on an FPGA. We systematically analyze the impact of applying SBQ both post-training and during the optimization loop (Quantization-Aware Training), and further explore the use of a custom, hardware-efficient function library designed to map directly to DSP block structures.

The complete pipeline was validated on the ‘pendulum-v1‘ benchmark. Our results demonstrate that while standard SINDy can accurately approximate the RL teacher policy, our hardware-oriented function library, struggles to capture the full complexity of the control task. This highlights a key trade-off between hardware-efficiency and model expressiveness. This work serves as a successful proof-of-concept and contributes novel techniques essential for deploying modern control algorithms on resource-constrained robotic systems.

Preface

This thesis marks the conclusion of our Bachelor studies in Electrical Engineering at Delft University of Technology. Over the course of this project, we had the opportunity to explore a multidisciplinary topic that challenged us to apply and expand our knowledge in areas such as control theory, reinforcement learning, and hardware-aware design.
Working together on this project allowed us to learn from each other's strengths, share responsibilities, and collaborate effectively throughout the process — from initial concept development to experimentation, analysis, and final documentation. It has been both technically demanding and personally rewarding.
We would like to express our sincere gratitude to our supervisor, Dr. Charlotte Frenkel, for her dedicated support, feedback, and guidance throughout the project. We are also especially thankful to Prof. dr. Guido de Croon for his valuable insights and encouragement, which helped us shape the direction of this research.
We would also like to thank Louka Verstraete and Nikolas Constantinou, who worked on the hardware architecture track of the same project. Although our theses were conducted independently, it was a pleasure to collaborate and exchange ideas throughout the process. Their input, shared discussions, and coordination contributed positively to both sides of the project and made the experience more engaging and insightful. Finally, we extend our heartfelt thanks to our friends and families for their ongoing support and motivation, not only during this thesis but throughout our academic journey.
We hope that this work provides a useful foundation for further research in interpretable, hardware-efficient control systems and contributes positively to the growing field of learning-based control.

Contents

Abstract i

Preface ii

1 Introduction 1
- 1.1 Problem Definition . . . 1
- 1.2 Hardware Implementation Target . . . 2
- 1.3 The Project's Aim . . . 2
- 1.4 Thesis Structure . . . 2

2 Background and Theory 3
- 2.1 Reinforcement Learning . . . 3
  - 2.1.1 Markov Decision Process . . . 3
  - 2.1.2 Value Functions and the Bellman Equation . . . 4
  - 2.1.3 Advanced Actor-Critic Algorithms . . . 5
- 2.2 Sparse Identification of Nonlinear Dynamics . . . 9
  - 2.2.1 SINDy with Control . . . 10
  - 2.2.2 PySINDy: A Python Implementation of SINDy . . . 10
  - 2.2.3 Policy Distillation with SINDy . . . 12

3 Programme of Requirements 13
- 3.1 General Requirements . . . 13
- 3.2 Software Subgroup Requirements . . . 14

4 Implementation 15
- 4.1 Framework and Workflow . . . 15
- 4.2 Creating the "teacher" policy . . . 15
  - 4.2.1 Environment . . . 16
  - 4.2.2 Policy Training and Selection . . . 16
  - 4.2.3 RL Results . . . 18
- 4.3 Creating the "student" policy . . . 20
  - 4.3.1 SINDy-based Approximation . . . 20
  - 4.3.2 Hardware-Efficient Function Library . . . 21
  - 4.3.3 Sparse Bit Quantization (SBQ) . . . 21
  - 4.3.4 SINDy results . . . 23
- 4.4 Final Results . . . 24

5 Recommendations and Future Work 26
- 5.1 Reinforcement Learning . . . 26
- 5.2 SINDy and Policy Distillation . . . 26
- 5.3 Hardware-Oriented Optimizations . . . 26
- 5.4 Pipeline Extension and Real-World Deployment . . . 26
- 5.5 Reconsideration of Requirements and Scope . . . 27

6 Conclusion 28

A Terminology 29

B Extra results **31**
B.1 Versions and seeds . . . 31
B.2 Additional SINDy equations for the pendulum . . . 32
B.3 MountainCarContinuous-v0 results . . . 33
B.3.1 Environment Description . . . 33
B.3.2 Results . . . 34
B.4 CartPole-v1 results . . . 40
B.4.1 Environment Description . . . 40
B.4.2 Results . . . 40

C Delfly environment **43**
C.1 Action space . . . 43
C.2 State Space . . . 43
C.3 Reset function . . . 43
C.4 Step function . . . 44
C.5 Render function . . . 44
C.6 Results . . . 44

D Project Repository **45**

E Code explanation **45**
E.1 RL . . . 45
E.1.1 train.py . . . 45
E.1.2 test.py . . . 45
E.1.3 gen_data_for_sindy.py . . . 45
E.2 SINDy . . . 45
E.2.1 create_sindy_policies.py . . . 45
E.2.2 compare_SINDY_vs_RL.py . . . 46
E.2.3 make_ready_for_dsp_flow.py . . . 46
E.3 Custom . . . 46
E.3.1 quantization.py . . . 46
E.3.2 CustomDynamicsEnv_v3.py . . . 46
E.3.3 custom_optimizers.py . . . 46
E.3.4 custom_libs.py . . . 47

F Requirements **48**
F.1 General Requirements . . . 48
F.2 Software Subgroup Requirements . . . 49

1 Introduction

1.1 Problem Definition

Flapping-wing micro air vehicles (FWMAVs), such as the DelFly shown in Figure 6, represent a frontier in aerial robotics. Their flight is governed by highly nonlinear and unsteady aerodynamics. This inherent complexity means that traditional linear controllers are often insufficient for achieving stable and agile flight, necessitating the use of more sophisticated control strategies[1].

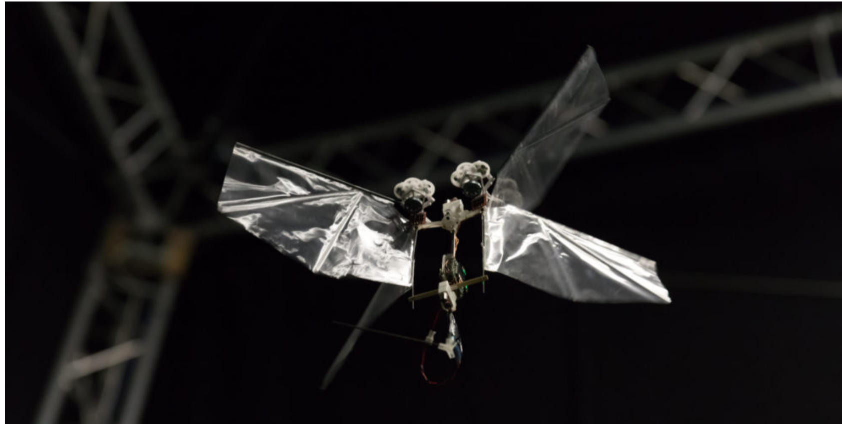

Figure 1: The DelFly Nimble, an agile flapping-wing micro air vehicle.

This leads to a fundamental conflict between performance and practicality. On one hand, a high-performance controller capable of handling these dynamics would be computationally complex. On the other hand, FWMAVs, such as the DelFly Nimble, are subject to extreme size, weight, and power constraints. With a total weight often under 30 grams and a power budget of only a few watts[2], the DelFly can only carry lightweight controllers with limited processing capability. This forces designers to sacrifice control quality for a simpler algorithm that fits within the drone's tight constraints.

It is desired to have a sophisticated controller able enough to handle the complex nonlinear dynamics of the FWMAV but also compact enough to satisfy the drones weight limits. One way of achieving this task is to create a complex non-linear controller and then sparsify it while keeping the core functionality. First a method for creating such a sophisticated controller must be chosen.
For a task of this complexity, two of the most powerful modern strategies are Model Predictive Control (MPC) [3] and Reinforcement Learning (RL)[4]. While MPC is a powerful technique that can handle system constraints, it requires solving a computationally intensive optimization problem at each time step, making it infeasible for real-time deployment on hardware with a power budget of only a few watts.
This leaves Reinforcement Learning as a highly promising alternative. RL approaches learn effective control policies through trial-and-error interaction, bypassing the need for a perfect analytical model and shifting the computational burden to an offline training phase. Recent advances have shown that deep RL agents can learn high-performance control policies even in the most challenging nonlinear domains [5], [6], [7], [8]. However, these policies rely on overparameterized neural networks that require millions of floating-

point operations for inference, making them difficult to interpret and, in their raw form, still too demanding for the DelFly's hardware [9].

For the sparsification, a recent breakthrough in data-driven modeling, the Sparse Identification of Nonlinear Dynamics (SINDy) algorithm [10] comes to mind. SINDy can distill a high-performance, complex control policy into a compact, symbolic equation.

1.2 Hardware Implementation Target

To implement the final, sparse controller, we target a Field-Programmable Gate Array (FPGA). Unlike traditional processors, FPGAs consist of a configurable fabric of logic blocks and interconnects that can be programmed to create custom digital circuits. This offers several key advantages for our project.
First, FPGAs make an ideal platform for research and iterative design due to their reconfigurability. Second, their inherent parallelism allows for the implementation of custom arithmetic operations that can be executed concurrently in a single clock cycle. This is particularly relevant for our hardware-oriented optimizations, which are designed to map directly onto the "Digital Signal Processing (DSP) blocks" within the FPGA. These specialized blocks are highly optimized for performing mathematical operations like multiplication and addition, and are a key resource we aim to use efficiently [11]. By designing a controller whose structure mirrors that of the DSP blocks, we can achieve significant gains in computational speed and power efficiency compared to a general-purpose processor.

1.3 The Project's Aim

Building upon previous work that combines Reinforcement Learning with SINDy [12], [13],[14], the central challenge of this project is to create an energy-efficient and robust controller suitable for resource-constrained hardware. The primary contribution of this thesis is to develop and evaluate a two-stage pipeline that derives a sparse, symbolic controller and optimizes it for hardware efficiency.
Our approach first leverages deep RL to train a high-performance "teacher" policy. Second, we use an enhanced SINDy framework to distill this policy into a simple, hardware-efficient "student" controller, which is further improved with hardware-aware techniques. We will validate this pipeline as a proof-of-concept for the DelFly on the classic "pendulum-v1" benchmark from OpenAI's Gymnasium python library[15].

1.4 Thesis Structure

This thesis is organized as follows. Chapter 2 provides the necessary background on the core methods, Reinforcement Learning and SINDy. Chapter 3 outlines the project requirements program. Chapter 4 discusses the implementation of our framework and presents the results. In Chapter 5, we reflect on the outcomes of this project, offering both recommendations and a discussion of promising directions for future work. Finally, Chapter 6 provides the conclusion and recommendations for future work.

2 Background and Theory

This chapter provides a detailed theoretical foundation for the methods used in this thesis. We begin with the fundamentals of Reinforcement Learning. We then introduce the SINDy framework, detailing the core algorithm, its optimizers, and our hardware-oriented extensions that are central to this work.

2.1 Reinforcement Learning

Reinforcement learning is a machine learning paradigm where an autonomous agent learns to make optimal decisions by interacting with an environment. Through a process of trial and error, the agent takes an action a_t in a given state s_t, and the environment responds with a numerical reward R_t and a new state s_{t+1}, as depicted in Figure 2. The goal is to learn a behavior, or policy, that maximizes the total reward accumulated over time. In this thesis, policy and controller are used interchangeably.

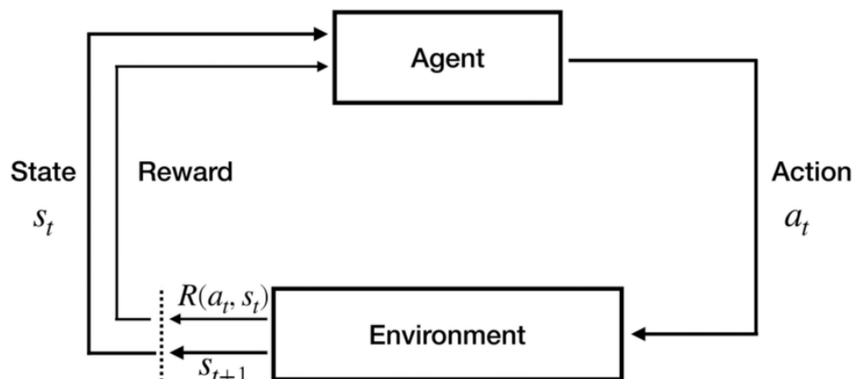

Figure 2: The general agent-environment interaction loop in reinforcement learning.

2.1.1 Markov Decision Process

This learning problem is formally framed as a "Markov Decision Process" (MDP). An MDP is a mathematical framework for modeling decision-making and is defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where:

- $\mathcal{S}$ is the set of all possible states.
- $\mathcal{A}$ is the set of all possible actions.
- $P(s'|s, a)$ is the transition probability of moving to state s'(s' when regarding continuous problems is equivalent to s_{t+1} when regarding discrete problems) from state s after taking action a.
- R(often also r or r_t) is the immediate reward received after taking action a in state s.
- $\gamma \in [0, 1]$ is the discount factor, which prioritizes immediate rewards over future ones, this will become clear further on.

The agent's strategy is called a policy, $\pi(a|s)$, which defines the probability of taking action a in state s. The objective within this framework is to find an optimal policy, π^*, that maximizes the expected discounted sum of future rewards, known as the return $G(\pi)$:

$$G(\pi) = E_\pi \left[\sum_{t=0}^{T} \gamma^t r_t \right]$$

The return is "discounted" by the factor γ. Since $0 \leq \gamma < 1$, multiplying a reward at time step t by γ^t gives it an exponentially smaller weight than a reward received at time t=0, encoding a preference for immediate rewards.

2.1.2 Value Functions and the Bellman Equation

To find an optimal policy, RL algorithms make use of "value functions", which estimate the long-term desirability of states or actions by making use of the return. The two primary value functions are:

- **State-value function $V^\pi(s)$:** The expected return the agent receives independent of which action it takes in state a
- **Action-value function $Q^\pi(s,a)$:** The average reward the agent receives when it takes an action in some state

These functions adhere to a fundamental recursive relationship known as the "Bellman expectation equation". This equation is crucial as it allows an agent to learn value functions iteratively. For the state-value function, it is defined as:

$$V^\pi(s) = E\left[R(s,a) + \gamma V^\pi(s')\right]$$

This equation states that the value of being in state s is the immediate reward plus the discounted value of the next state, averaged over all possible actions and transitions. The expectation E here is taken over two sources of randomness: The agent's action a, which is chosen according to its policy $\pi(a|s)$ and the environment's next state s', which occurs according to the transition probability $P(s'|s,a)$.
By expanding this expectation, we get the full Bellman equation, which explicitly averages over these probabilities:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \right)$$

Similarly, the action-value function, which gives the value of taking a specific action a in state s, is defined as:

$$Q^\pi(s,a) = E\left[R(s,a) + \gamma V^\pi(s')\right]$$

In this case, since the action a is already chosen, the expectation is only over the possible next states s' determined by the environment's transition probability $P(s'|s,a)$. Expanding this expectation yields:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s')$$

By substituting the definition of $V^\pi(s')$ into the equation above, we can express the Q-function entirely in terms of Q-values, which is common in many algorithms:

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s',a')$$

This recursive relationship is fundamental to the operation of actor-critic methods. In this paradigm, both the policy and the value function are represented by deep neural networks, which serve as function approximators capable of handling large, continuous state and action spaces. The architecture consists of two distinct components: the "critic", which learns a value function (e.g., Q^π) by leveraging the Bellman equation, and the "actor", which represents the policy and is refined based on the evaluative feedback from the critic.

2.1.3 Advanced Actor-Critic Algorithms

For our control task, we utilize two state-of-the-art actor-critic algorithms:

Proximal Policy Optimization :
Proximal Policy Optimization (PPO) is an "on-policy" algorithm prized for its stability[16]. On-policy means the algorithm learns exclusively from data generated by the most recent version of its policy. PPO's core principle is to take cautious policy updates by maximizing a "clipped" objective function, which prevents large, potentially destabilizing changes:

$$L^{\text{CLIP}}(\theta) = E_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

To understand this objective, we first define its two main components.

The Advantage Function ($\hat{A}_t$) The advantage function answers the question: "How much more reward did I receive from the action I just took compared to the average reward I usually receive in this state?" It's formally defined as $\hat{A}_t = Q(s_t, a_t) - V(s_t)$. Remebering the previously stated definitions, this can be read as the difference between the expected return of a particular action in a state and the average return you get when being in that state. A positive advantage indicates a better-than-average action to be encouraged, while a negative advantage indicates a worse-than-average action to be discouraged. Using the advantage provides a baseline for comparison, which reduces variance and stabilizes training.

The Probability Ratio ($r_t(\theta)$) The probability ratio measures how likely the new, updated policy (π_θ) is to select an action compared to the old policy ($\pi_{\theta_{old}}$) that collected the data. During data collection, the policy is frozen to ensure consistency, and it remains unchanged for a fixed number of gradient steps while the optimizer updates the policy based on the collected batch. The probability ratio is defined as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. A ratio greater than 1 means the action is now more likely.

The Clipped Objective A simpler policy gradient objective would be to simply maximize $r_t(\theta)\hat{A}_t$ as this would increase the likelihood of a certain action when the advantage is positive or decrease the likelihood when the advantage is negative. The problem is that a very large advantage could cause an excessively large update to the policy, destabilizing the learning process.
PPO solves this by "clipping" the objective. The $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$ function constrains the probability ratio to stay within a small window around 1 (depending on ϵ, e.g., $[0.8, 1.2]$). The 'min' function then creates a pessimistic bound.
When the advantage is positive ($\hat{A}_t > 0$): The 'min' function selects the smaller of the normal objective and the clipped objective. This puts a cap on the potential reward from a single update, preventing over-optimism and an overly large step.
When the advantage is negative ($\hat{A}_t < 0$): The 'min' function's behavior effectively chooses the maximum (least negative) of the two terms. This limits how much the policy can change in response to a single bad action, preventing it from over-correcting too aggressively.
In short, the clipped objective creates a "trust region" around the old policy. It allows for small, safe updates but punishes the agent for trying to change its policy too drastically, which is the key to PPO's stability.

Soft Actor-Critic :
Soft Actor-Critic (SAC) is a highly sample-efficient "off-policy" actor-critic algorithm based on the maximum entropy framework [17]. Being "off-policy" allows it to learn from a replay buffer of past experiences, significantly improving data efficiency.
The core idea of SAC is to augment the standard RL objective of maximizing rewards with an entropy maximization term. The agent is, therefore, trained to succeed at the task while acting as randomly as possible. This encourages exploration and prevents the policy from converging prematurely to a suboptimal strategy. The objective function is:

$$J(\pi) = \sum_{t=0}^{T} E\left[R(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))\right]$$

where $\mathcal{H}$ is the policy entropy and the temperature parameter α controls the trade-off between reward maximization (exploitation) and entropy maximization (exploration).
To optimize this objective, SAC uses a sophisticated actor-critic architecture:

One Actor: A policy network π_ϕ that learns to output actions.

Two Critics: Two separate Q-networks, Q_{θ_1} and Q_{θ_2}, are trained to estimate the action-value function. This technique, known as Clipped Double Q-Learning, mitigates the tendency of Q-learning to overestimate values, leading to more stable training.

The critics are trained by minimizing the soft Bellman residual error. The loss for each critic is:

$$J_Q(\theta_i) = E\left[\frac{1}{2}\left(Q_{\theta_i}(s_t, a_t) - y\right)^2\right]$$

where the target value y incorporates the entropy term and uses the minimum of the two

critic predictions for the next state to provide a conservative Q-value estimate:

$$y = r_t + \gamma \left(\min_{i=1,2} Q_{\bar{\theta}_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\phi(a_{t+1}|s_{t+1}) \right), \quad a_{t+1} \sim \pi_\phi$$

The crucial component here is the term $-\alpha \log \pi_\phi(a_{t+1}|s_{t+1})$, which is directly related to the policy's entropy. The log-probability, $\log \pi$, is a large negative number for actions that are highly uncertain or random. By subtracting this term, the algorithm adds a positive "entropy bonus" to the target value. This effectively acts as an intrinsic reward for exploration, encouraging the actor to try less predictable actions. The target y thus represents not just the future environmental reward (from the Q-value), but the reward plus a bonus for maintaining randomness. This helps prevent the policy from collapsing into a deterministic, suboptimal strategy and improves the robustness of the final controller.
The actor network, π_ϕ, is then updated by minimizing its own loss function. This objective is designed to encourage the actor to select actions that have high Q-values as estimated by the critics, while also maintaining policy entropy. The loss is given by:

$$J_\pi(\phi) = E \left[\alpha \log(\pi_\phi(a_t|s_t)) - \min_{i=1,2} Q_{\theta_i}(s_t, a_t) \right], \quad a_t \sim \pi_\phi$$

This loss function creates a trade-off between two competing goals:
On the one hand there is exploitation; to minimize the term $-\min Q_{\theta_i}(s_t, a_t)$, the actor must learn to choose actions a_t that result in the highest possible Q-value. This component pushes the policy to exploit its current knowledge to maximize rewards.
On the other hand there is exploration; to minimize the term $\alpha \log(\pi_\phi(a_t|s_t))$, the actor is encouraged to make $\log(\pi_\phi)$ as negative as possible. The log-probability is more negative for less likely actions. This component, therefore, pushes the policy to be more stochastic, which aids exploration and prevents premature convergence to a suboptimal solution.
The temperature parameter α controls the balance between these two objectives. By minimizing this combined loss, the actor learns a policy that is both effective at solving the task and robustly exploratory.
The combination of off-policy learning, entropy maximization, and clipped double Q-learning makes SAC one of the most powerful and sample-efficient algorithms for continuous control problems.

Twin Delayed Deep Deterministic Policy Gradient :
Twin Delayed Deep Deterministic Policy Gradient (TD3) is another powerful off-policy actor-critic algorithm that specifically addresses the function approximation errors common in actor-critic methods [18]. The primary failure mode it tackles is the overestimation of Q-values, where the critic provides overly optimistic value estimates that lead the actor to learn a faulty policy. As seen before with PPO and SAC, the mitigating of this problem is a common trend in modern RL algorithms. TD3 makes use of three key techniques to tackle this problem:

Clipped Double Q-Learning: TD3 learns two critic networks (similar to PPO and SAC) and uses the minimum of their predictions when forming the Bellman target. This helps to reduce overestimation bias. The target y is calculated as:

$$y = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi'}(s') + \epsilon)$$

Here, the target y consists of the immediate reward plus the discounted value from the Clipped Double Q-Learning technique, which uses a noise-smoothed action from the target policy. The purpose and mechanics of this will be elaborated upon further.

Target Policy Smoothing: When calculating the target value, TD3 adds a small amount of clipped noise ϵ to the action selected by the target actor ($\pi_{\phi'}$). This smooths the value estimate along changes in actions, making the policy more robust to errors.

Delayed Policy Updates: The actor network (π_{ϕ}) is updated less frequently than the critic networks. This allows the critic's value estimate to converge before being used to guide the actor, leading to higher-quality policy updates.
Together, these three actions make TD3 a strong baseline algorithm for continuous control tasks.

2.2 Sparse Identification of Nonlinear Dynamics

SINDy is a data-driven method for discovering simple, governing equations from measurement data. The measurement data are sampled from the governing unknown, usually non-linear, dynamics :

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

SINDy operates on the principle of sparsity, which assumes that most complex systems can be described by only a few key dynamic terms. The process begins by constructing a library of candidate functions, $\Theta(\mathbf{x}) \in R^{N\times M}$, where N is the number of data points and M is the amount of candidate functions in the library. Given data of a system's state $\mathbf{x} \in R^{N\times D}$ where D are the number of states, and its derivative $\dot{\mathbf{x}} \in R^{N\times D}$, SINDy finds a sparse set of coefficients $\Xi \in R^{M\times D}$, such that:

$$\dot{\mathbf{x}} \approx \Theta(\mathbf{x})\Xi$$

Thus SINDy finds a combination of the candidate functions in the library that approximates the state-derivatives. This is not too distinct from other symbolic regression methods. However, the key innovation is to find the **sparsest** possible solution for Ξ. This objective can be expressed as minimizing the number of nonzero elements, denoted by the $\|\cdot\|_0$ norm, while keeping the model error below a tolerance ϵ:

$$\min_{\Xi} \|\Xi\|_0 \quad \text{subject to} \quad \|\dot{\mathbf{x}} - \Theta(\mathbf{x})\Xi\|_2 < \epsilon$$

This is solved using sparsity enforcing optimizers, such as Sequentially Thresholded Least Squares (STLSQ) or Sparse Relaxed Regularized Regression (SR3), which will be elaborated on later. Figure 3 illustrates this process for the classic Lorenz system, where the dynamics equations can be seen in the top left of the figure. SINDy successfully discovers the underlying differential equations from trajectory data alone.

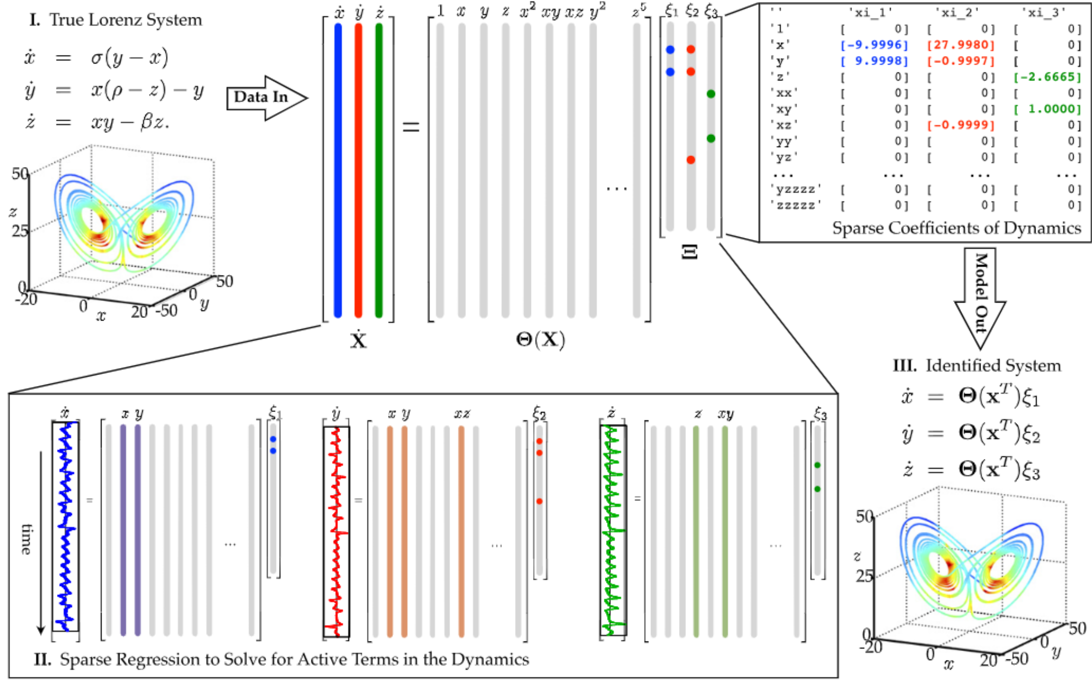

Figure 3: An example of SINDy applied to the Lorenz system. From time-series data, a candidate library is constructed, and sparse regression identifies the few active terms that correctly describe the system's dynamics.[10]

2.2.1 SINDy with Control

In many real-world applications, systems are influenced not only by their internal state but also by external control inputs. SINDy with Control (SINDy-C) extends the original SINDy framework to account for these control signals[19].
The system dynamics now include a control input $\mathbf{u}(t) \in R^Q$:

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

This leads to the extended regression formulation:

$$\dot{\mathbf{X}} \approx \Theta(\mathbf{X}, \mathbf{U})\Xi$$

Where:

- $\mathbf{U} \in R^{N\times Q}$ is the matrix of control inputs.
- $\Theta(\mathbf{X}, \mathbf{U})$ is a library of candidate functions involving both the state and control variables (e.g., x, u, xu, x^2u).

By including $\mathbf{u}(t)$, SINDy-C enables model identification for systems with actuation, which is crucial for control design, simulation, and prediction in engineering systems. Note that for SINDy-C the control values need to be recorded so that candidate functions can be made from the data.

2.2.2 PySINDy: A Python Implementation of SINDy

`PySINDy` is an open-source Python library that implements the SINDy framework[20], including support for:

- Building libraries of candidate functions (polynomials, Fourier terms, custom functions).
- Estimating derivatives using finite difference methods or total variation regularization.
- Solving sparse regression problems using various optimizers.
- Extensions such as SINDy-C, SINDy-PI (for implicit systems), and more.

To use PySINDy, the user provides:

- Time-series data of the state $\mathbf{x}(t)$ and optionally control inputs $\mathbf{u}(t)$.
- A choice of function library, such as polynomial or custom libraries.
- A sparse optimizer.

Sparse Optimizers in PySINDy

Several optimizers are available in PySINDy to enforce sparsity in the coefficient matrix Ξ:

- **Sequentially Thresholded Least Squares (STLSQ)**:
  STLSQ[21] begins by solving a standard least-squares problem:

  $$\min_{\Xi} \|\dot{\mathbf{X}} - \Theta(\mathbf{X})\Xi\|_2^2$$

  After obtaining an initial estimate of Ξ, it applies a threshold to zero out coefficients with magnitudes below a specified value. The algorithm then solves the least-squares problem again on this reduced system, using only the indices of the non-zero coefficients. This process is repeated iteratively until the solution converges:

  1. Solve the least squares to estimate Ξ.
  2. Set coefficients with magnitude less than a threshold τ to zero.
  3. Refit the model using only nonzero coefficients.
  4. Repeat until convergence.

  This method is fast and effective in low-noise regimes, but its performance can degrade with noisy data or correlated features.

- **Lasso**:
  Lasso (L1 regularization)[22] augments the least-squares problem with an L_1 penalty to promote sparsity:

  $$\min_{\Xi} \|\dot{\mathbf{X}} - \Theta(\mathbf{X})\Xi\|_2^2 + \lambda\|\Xi\|_1$$

  where λ is a regularization parameter controlling the strength of sparsity. The L_1 norm encourages many coefficients in Ξ to be exactly zero. Lasso is convex and effective for high-dimensional problems, but may struggle with correlated features.

- **SR3**:
SR3 (Sparse Relaxed Regularized Regression)[23] introduces an auxiliary variable $\mathbf{Z}$ to decouple the regression from the sparsity regularization. Instead of enforcing sparsity directly on Ξ, SR3 minimizes:

$$\min_{\Xi,\mathbf{Z}} \|\dot{\mathbf{X}} - \Theta(\mathbf{X})\Xi\|_2^2 + \lambda\|\mathbf{Z}\|_1 + \nu\|\Xi - \mathbf{Z}\|_2^2$$

Here:

  - Ξ is responsible for fitting the data,
  - $\mathbf{Z}$ acts as a sparse surrogate for Ξ,
  - λ controls the sparsity level of $\mathbf{Z}$,
  - ν controls how tightly Ξ is pulled toward $\mathbf{Z}$.

This relaxation provides more flexibility than methods like Lasso. By separating data fitting and sparsity enforcement, SR3 often recovers better models in the presence of noise or correlated features. It can be seen as interpolating between Ridge regression (when ν is small) and Lasso (when ν is large and $\Xi \approx \mathbf{Z}$).

- **Constrained SR3**:
Constrained SR3 replaces the soft penalty on $\|\Xi - \mathbf{Z}\|_2^2$ with a strict constraint. The optimization becomes:

$$\min_{\Xi,\mathbf{Z}} \|\dot{\mathbf{X}} - \Theta(\mathbf{X})\Xi\|_2^2 + \lambda\|\mathbf{Z}\|_1 \quad \text{subject to} \quad \|\Xi - \mathbf{Z}\|_2^2 \leq \epsilon$$

This formulation explicitly restricts how far the fitted model Ξ can deviate from the sparse structure encoded in $\mathbf{Z}$. The constraint ϵ allows practitioners to precisely control the trade-off between model accuracy and sparsity, which is useful in applications requiring strong guarantees on interpretability or robustness. In constrained SR3, it is possible to impose explicit constraints on the coefficients. For example, one can enforce relationships such as $c_1 = -c_2$, $c_1 + c_2 = 10$, or bounds like $|c_1| < 10$. This feature is particularly useful when optimizing for hardware deployment, as it allows us to constrain the solution such that all coefficients remain within the range representable by a given fixed-point bitwidth.

Each optimizer has its own trade-offs in terms of computational cost, sparsity, and robustness to noise. The choice of optimizer can affect which terms appear in the final model, especially when data is noisy or limited.

2.2.3 Policy Distillation with SINDy

While SINDy was designed for discovering physical dynamics, it can also be repurposed for policy distillation. For example, instead of using state derivative data, state-action pairs $(\mathbf{s}, \mathbf{a})$ generated by a trained RL teacher policy are used. The SINDy objective then becomes finding a sparse function that approximates the policy:

$$\mathbf{a}(\mathbf{s}) \approx \Theta(\mathbf{s})\Xi$$

This allows for the distillation of a complex, "black-box" neural network into a compact, interpretable formula ideal for hardware implementation.

3 Programme of Requirements

To guide the development process and define clear evaluation criteria, a structured set of requirements has been established. These requirements outline the intended functionality, performance goals, and design constraints for the system. They are categorized into general project-level requirements and specific requirements for the software and algorithm subgroup. To provide context for the requirements detailed in this chapter, a high-level overview of the entire project pipeline is presented in Figure 4. This overview illustrates how the project's main stages are interconnected and clarifies the division of responsibilities between the software and hardware subgroups.
The pipeline consists of three primary stages:

1. **Reinforcement Learning Stage:** The process is initiated within a simulated environment, where a high-performance neural network, designated as the "teacher" policy, is trained. The methodology is first validated on the pendulum-v1 proof-of-concept environment, with the ultimate application being a DelFly drone simulation.

2. **Policy Distillation Stage:** The complex teacher policy is then processed by the SINDy-based distillation framework. This stage, which is the core focus of the software subgroup, produces a sparse and computationally lightweight symbolic controller (the "student" policy).

3. **Hardware Implementation Stage:** Finally, the simplified symbolic controller is given to the hardware subgroup, who are responsible for translating this mathematical expression into an optimized hardware description and producing the final FPGA design.

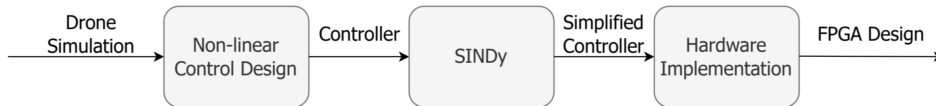

Figure 4: A high-level overview of the complete project pipeline, from RL policy training to final FPGA design.

3.1 General Requirements

These requirements define the high-level goals for the entire project, encompassing both the software and hardware development efforts and are therefore joint with the hardware implementation subgroup.

- The system, as a baseline, must use a neural-network based controller to control an agent in a simulation environment.

- The system must use SINDy to sparsify the neural network-based controller and provide a simplified mathematical expression representing the controller.

- The system must be able to convert the simplified controller into an FPGA design.

- The system's efficacy must be demonstrated for the case of a pendulum simulation environment as a proof-of-concept.

3.2 Software Subgroup Requirements

These requirements outline the specific tasks, performance goals, and evaluation criteria for the software subgroup. They focus on the development and validation of the RL-to-SINDy policy distillation pipeline.

1. The trained RL controller must achieve a mean episodic reward sufficient to be considered a successful solution for the given control task.
2. The distilled SINDy policy must achieve performance (mean episode rewards) comparable to the original RL policy that it approximates.
3. A dedicated hardware-oriented SINDy optimizer shall be developed and evaluated, with its performance compared against baseline optimizers.
4. The SINDy distillation process must produce a sparse policy, defined by a minimal number of active terms, suitable for efficient hardware implementation.
5. The effect of implementing and using a hardware-efficient function library within the SINDy framework must be evaluated in terms of final policy performance (mean episode rewards).
6. The learned RL and distilled SINDy policies must demonstrate robustness by successfully stabilizing the system from a wide range of initial conditions.

4 Implementation

This chapter details the practical implementation of the proposed pipeline for developing a hardware-efficient controller. The framework is developed in Python within a Visual Studio Code environment; all versions of the relevant libraries and Python used for this thesis are located in Appendix B. The core methodologies are based on established machine learning libraries. The following sections describe the structure of the software framework, the process for training the RL "teacher" policy, and the subsequent distillation and optimization of the SINDy-based "student" controller, focusing on the 'pendulum-v1' environment as a proof-of-concept.

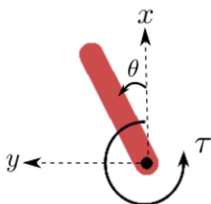

Figure 5: The Pendulum-v1 environment from Gymnasium.

4.1 Framework and Workflow

The general workflow is shown in figure 6, and proceeds as follows:

Figure 6: overview of the software algorithm

- **RL Policy Training:** A high-performance neural network policy (the "teacher") is trained to solve the control task using state-of-the-art RL algorithms.
- **Data Generation:** The trained teacher policy is used to generate a dataset of state-action pairs.
- **SINDy Policy Distillation:** The dataset is used to train and identify multiple sparse, symbolic "student" policies using various SINDy configurations, including our hardware-oriented additions.
- **Comparative Analysis:** The performance of the SINDy student policies is benchmarked against the original RL teacher policy.

4.2 Creating the "teacher" policy

In this section the methods for creating a high-performance controller using deep reinforcement learning will be discussed.

4.2.1 Environment

For environment simulation, we use **Gymnasium**[15], as it is the industry standard for developing and comparing RL algorithms. Its widespread support and standardized API ensure reproducibility and compatibility with a rich ecosystem of tools.
To validate our pipeline as a robust proof-of-concept, we tested our approach on several classic control environments, including ‘Pendulum-v1‘, ’MountainCarContinuous-v0’ and ’CartPole-v1’. While the pipeline was validated across several environments, this chapter will focus on the results from Pendulum-v1 as the primary case study, as it serves as the most representative proxy for the core challenges of DelFly attitude control. The results for the other environments can be found in Appendix B. The pendulum’s continuous torque input mirrors the continuous control required for the DelFly’s actuators. Furthermore, its core task of stabilizing an inherently unstable, nonlinear rotational system is a close analogue to the DelFly’s flight dynamics.

The Pendulum-v1 Environment The ‘Pendulum-v1‘ environment is a classic control problem where the goal is to swing a simple pendulum so that it stays upright, and to do so with minimal effort. This task is representative of many real-world control challenges that involve balancing an unstable system. The environment is mainly described by the following attributes:

- **Observation Space:** The state of the system is described by three variables: the cosine of the angle $\cos(\theta)$, the sine of the angle $\sin(\theta)$, and the angular velocity $\dot{\theta}$. The angle θ is zero when the pendulum is pointing straight up.
- **Action Space:** The agent can apply a continuous torque value ranging from -2.0 to +2.0 to the pendulum’s joint.
- **Reward Function:** The reward at each step is calculated based on the pendulum’s angle, its angular velocity, and the amount of torque applied. The agent is rewarded for keeping the pendulum upright ($\theta \approx 0$) with low velocity, and is penalized for using a large amount of torque. The exact formula is: $r = -(\theta^2 + 0.1 \times \dot{\theta}^2 + 0.001 \times \tau^2)$.

To improve training stability and performance, the environment is wrapped using ‘VectorizedEnv‘ for parallel data collection and observation normalization to scale the input features.

4.2.2 Policy Training and Selection

For the training of the RL agent we chose to benchmark several algorithms, including ”PPO, SAC, and TD3”, as they represent the current state-of-the-art in continuous control and are featured prominently in the literature we build upon. A grid search is performed over a predefined set of hyperparameters specific to each algorithm and can be seen in table 1. The hyperparameters not shown in the table were set to their default values.

Table 1: Grid search hyperparameter space for each RL algorithm.

Algorithm	**Parameter**	**Values**
PPO	learning_rate	{1e-3, 1e-4, 1e-5}
	batch_size	{256}
	n_steps	{1024}
	ent_coef	{0.001, 0.01}
SAC	learning_rate	{1e-3, 1e-4, 1e-5}
	buffer_size	{500,000}
	tau	{0.005, 0.01}
	batch_size	{256, 512}
TD3	learning_rate	{1e-3, 1e-4, 1e-5}
	buffer_size	{500,000}
	tau	{0.005, 0.01}
	batch_size	{256, 512}

The training algorithms are implemented using the robust models provided by the Stable-Baselines3[24] library. During the hyperparameter search, key performance metrics are logged using TensorBoard [25], an interactive visualization toolkit that allows for real-time monitoring of the learning curves. The full selection procedure is formalized in Algorithm 1. As shown in the algorithm, the process begins with a grid search performed independently for each RL algorithm. During this phase, numerous models are trained with different hyperparameter configurations. From each grid search, the single model with the best performance is selected as a candidate. These candidate models then proceed to a final, rigorous evaluation. In this final stage, each candidate is tested over 10 separate episodes to calculate a stable mean reward. The model that achieves the highest mean reward across these evaluation trials is designated as the definitive teacher policy for the subsequent distillation stage.

Algorithm 1 Finding the Best Neural Network Teacher Policy

Require: List of algorithms to test, $\mathcal{A} = \{\text{PPO, SAC, TD3}\}$
Require: Set of hyperparameters for grid search, $\mathcal{P}$
Require: Number of evaluation trials, $N_{trials} = 10$

```
 1: procedure FindTeacherPolicy(A, P, N_trials)
 2:     candidate_models ← [ ]
 3:     best_teacher_policy ← None
 4:     max_reward ← −∞
                                          ▷ Stage 1: Grid Search for each algorithm
 5:     for each algo in A do
 6:         best_model_for_algo ← GridSearch(algo, P)
 7:         Append best_model_for_algo to candidate_models
 8:     end for
                                          ▷ Stage 2: Final Evaluation and Selection
 9:     for each model in candidate_models do
10:         rewards ← [ ]
11:         for i = 1 to N_trials do
12:             episode_reward ← EvaluatePolicy(model)
13:             Append episode_reward to rewards
14:         end for
15:         mean_reward ← Average(rewards)
16:         if mean_reward > max_reward then
17:             max_reward ← mean_reward
18:             best_teacher_policy ← model
19:         end if
20:     end for
21:     return best_teacher_policy
22: end procedure
```

4.2.3 RL Results

The results of the RL stage are presented here, beginning with a visual comparison of the learning curves from the best model found for each algorithm during its respective grid search. Figure 7 plots the mean evaluation reward against the number of training steps. The final hyperparameter configuartions for each algorithm are shown in table 2.

Hyperparameter	**PPO**	**SAC**	**TD3**
`learning_rate`	0.001	0.0001	0.001
`gamma`	0.99	0.99	0.99
`batch_size`	256	512	256
`n_steps`	1024	–	–
`ent_coef`	0.001	–	–
`buffer_size`	–	500000	500000
`tau`	–	0.01	0.005
`gradient_steps`	–	1	1
`train_freq`	–	1 step	–

Table 2: Hyperparameter configurations for PPO, SAC, and TD3 algorithms.

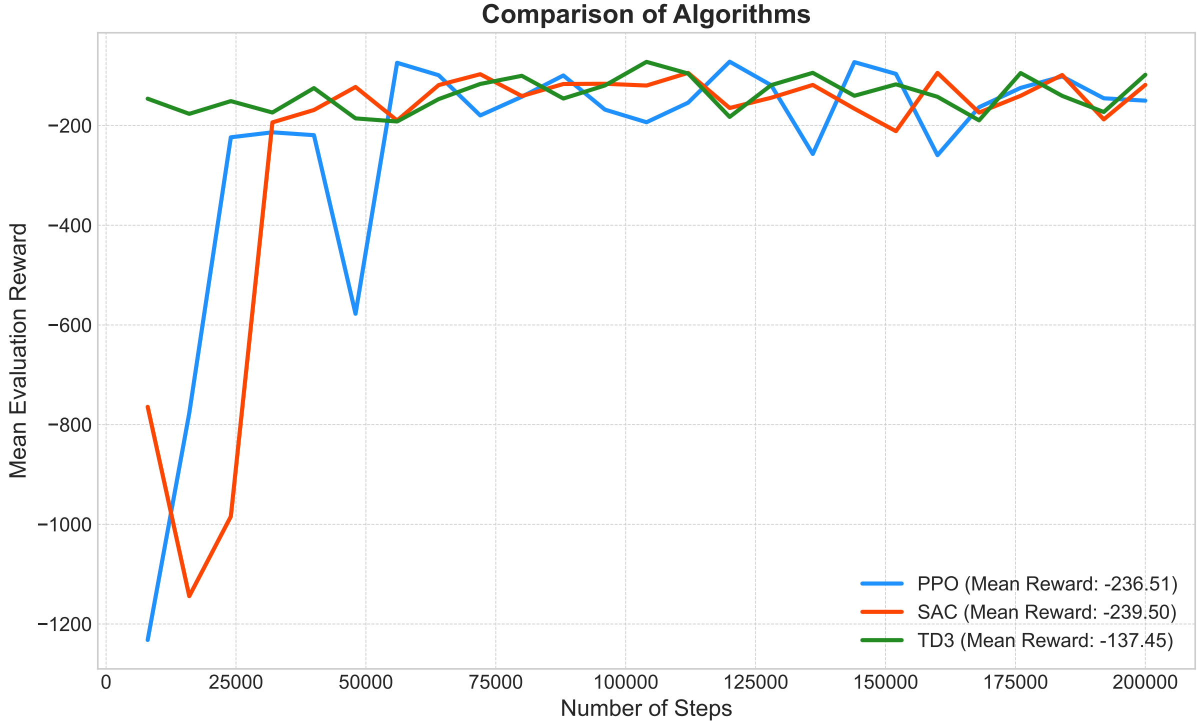

Figure 7: A comparison of the learning curves for the best-performing PPO, SAC, and TD3 models identified during the hyperparameter grid search.

As observed in the figure, the TD3 algorithm appears to learn significantly faster than the others, reaching a high reward plateau very early in the training process. However, a single training run can be misleading due to the stochastic nature of reinforcement learning. To make a more robust decision, the three candidate policies were subjected to a final evaluation over 10 episodes each. The resulting mean reward and standard deviation are presented in Table 3.

Table 3: Mean episodic reward and standard deviation for each algorithm after testing for 10 episodes.

Algorithm	**Mean Reward**	**Standard Deviation**
PPO	−119.46	74.52
SAC	−155.84	50.14
TD3	−155.39	50.22

The results of this more rigorous evaluation confirm that relying solely on the initial grid search would have been premature. Despite its rapid initial learning, the TD3 policy did not achieve the highest final performance. The PPO policy, with a mean reward of −119.46, outperformed the other candidates. Therefore, the best-performing PPO model was selected as the definitive ”teacher” policy for the subsequent SINDy distillation stage of the pipeline.

4.3 Creating the "student" policy

Once a teacher policy is trained, the goal is to distill its complex, "black-box" behavior into a simple, hardware-efficient symbolic model.

4.3.1 SINDy-based Approximation

The overall procedure for finding the best "student" policy is formalized in Algorithm 2. This process involves a comprehensive grid search over various SINDy configurations as shown in table 4.

Table 4: Hyperparameter search space for SINDy grid search.

Hyperparameter	**Search Range**
Threshold (STLSQ)	Logarithmic scale from 10^{-5} to 10^{2} (10 values)
Constraint (ConstrainedSR3)	Logarithmic scale from 10^{1} to 10^{10} (5 values)
α (regularization term)	Logarithmic scale from 10^{-5} to 10^{5} (10 values)

Where the hyperparameters control the following:

- For the **STLSQ** optimizer, the `threshold` parameter sets the magnitude below which coefficients are pruned from the model.
- For the **ConstrainedSR3** optimizer, the `threshold` acts as a constraint on the solution space.
- For both optimizers, the `alpha` (α) parameter is a regularization coefficient that controls the strength of the sparsity penalty, with higher values encouraging sparser models.

This search was conducted across all combinations of optimizers, function libraries, and our custom hardware-aware additions. The best model for each variant was then selected based on a single criterion: achieving the lowest Mean Squared Error (MSE) when compared to the output of the RL teacher policy.

Algorithm 2 Finding the Best SINDy Student Policy via Grid Search

Require: State-action data from teacher policy, $\mathcal{D} = \{(\mathbf{s}_i, \mathbf{a}_i)\}_{i=1}^{N}$
Require: Set of SINDy variants, $\mathcal{V}$ (e.g., optimizers, libraries, quantization schemes)
Require: Set of hyperparameters for grid search, $\mathcal{P}$

```
 1: procedure FindStudentPolicy(D, V, P)
 2:     best_model ← None
 3:     min_mse ← ∞
 4:     for each variant in V do
 5:         for each params in P do
 6:             model ← InitializeSINDy(variant, params)
 7:             model.fit(s, a)
 8:             current_mse ← CalculateMSE(model, D)
 9:             if current_mse < min_mse then
10:                 min_mse ← current_mse
11:                 best_model ← model
12:             end if
13:         end for
14:     end for
15:     return best_model
16: end procedure
```

4.3.2 Hardware-Efficient Function Library

To improve hardware performance, a specialized function library was created. It includes expressions that map directly to the structure of a DSP block, which typically contains a pre-adder, a multiplier, and a post-adder. A typical efficient expression is of the form $(A+B)C+D$, which can be computed in a single clock cycle. By building expressions entirely in this format, the resulting model becomes highly efficient for FPGA implementation. However, this structure introduces optimization challenges, as some library functions can be highly correlated, leading to potential numerical instability. Furthermore, since each composite function is multiplied by a single coefficient, the expressiveness of the model is restricted. For example, an expression like $c_1 \cdot AC + \frac{c_1}{2} \cdot BC$ cannot be represented, as SINDy would enforce a single coefficient for the entire term: $c_1(A+B)C = c_1AC + c_1BC$.

4.3.3 Sparse Bit Quantization (SBQ)

To address the computational bottleneck of multiplications in hardware, we propose an alternative to standard linear quantization. Instead of uniform steps, our Sparse Bit Quantization (SBQ) utilizes a non-uniform set of values constructed from combinations of one or two powers of two. This allows computationally expensive multiplications to be replaced by simple bit-shift and addition operations. For instance, a coefficient of 3.75 can be decomposed as $2^2 - 2^{-2}$, allowing the multiplication $3.75 \times x$ to be implemented as a sequence of two bit-shifts and a subtraction.
In our implementation, we explore two strategies for applying SBQ: as a post-processing step after model identification (post-training quantization) and integrated directly into the SINDy optimization loop (quantization-aware training).

Quantization Rules The set of allowable positive values in SBQ is defined by the union of the following five rules, where m is the number of integer bits and n is the number of fractional bits:

1. $\{2^i\}_{i=0}^{m}$
2. $\{2^{-j}\}_{j=0}^{n}$
3. $\{2^i \pm 2^{-j}\}_{i=0,j=0}^{m,n}$
4. $\{2^i \pm 2^k\}_{k=0,i>k}^{m}$
5. $\{2^{-j} \pm 2^{-l}\}_{l=0,j>l}^{n}$

These rules correspond to values that are efficient to compute in hardware. Rules 1 and 2 require a single bit-shift, while Rules 3, 4, and 5 require two bit-shifts and an addition/subtraction, which can often be executed in just two clock cycles. Figure 8 compares the resulting non-uniform SBQ grid with a traditional linear quantization grid.

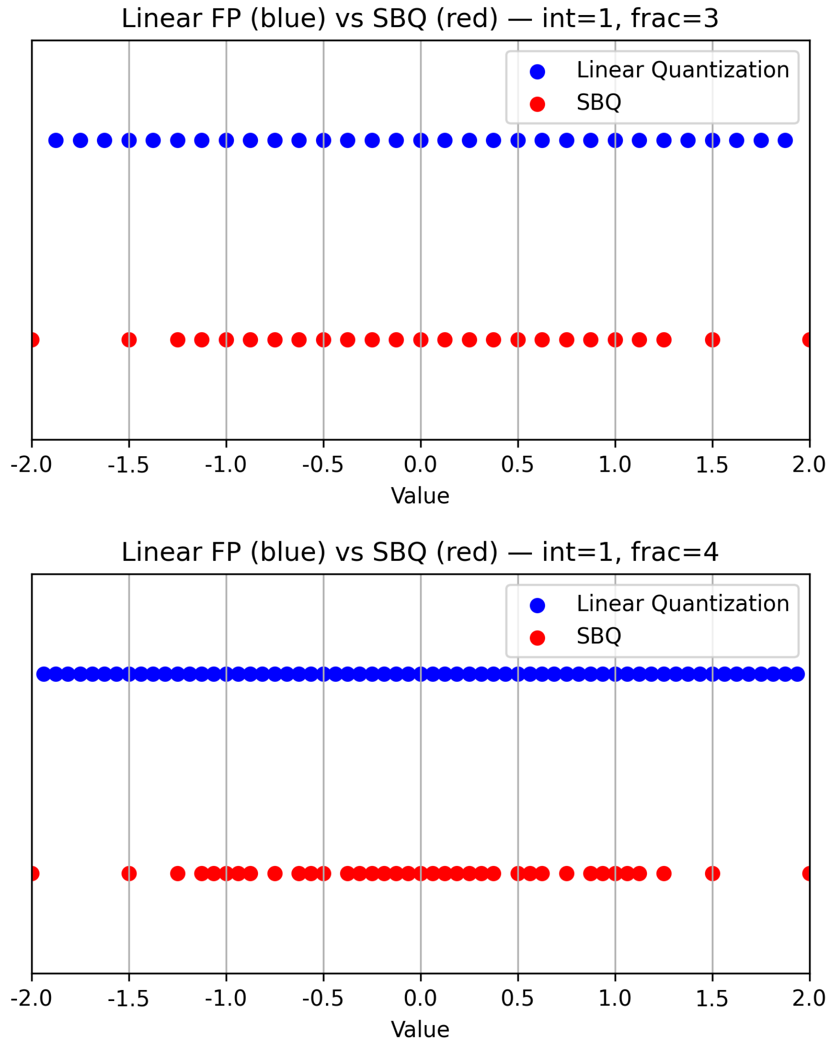

Figure 8: Quantized values using SBQ and linear quantization for 1 integer bit and 3 (top) or 4 (bottom) fractional bits.

Analysis of the SBQ plots reveals a non-uniform quantization grid. Increasing the number of fractional bits enhances the quantization resolution primarily around integer powers of two. Conversely, values generated by integer-only combinations, such as 2.0, remain isolated points. This highlights the inherent trade-off in SBQ between computational efficiency and uniform numerical resolution.

4.3.4 SINDy results

As stated before, each of the eight SINDy variants—spanning two function libraries, two optimizers, and two quantization approaches (with and without QAT)—was subjected to a grid search to find its optimal hyperparameters. The performance of the best model from each variant was then evaluated based on three key metrics:

- **MSE**: The Mean Squared Error between the SINDy policy's output and the teacher policy's output, before quantization. This measures the raw accuracy of the approximation.
- **PTQ-MSE**: The MSE after applying post-training quantization to the model's coefficients. This measures the performance degradation due to quantization. For this work, a bitwidth of 10 was used (4 integer bits, 6 fractional bits).
- **Nonzero Count**: The number of active terms in the final symbolic equation, which is a direct measure of the model's sparsity.

The results of this comprehensive search are summarized in Table 5, which also lists the final hyperparameters for each best-in-class model.

Table 5: Grid search results for the eight SINDy model variants.

Library	QAT	Optimizer	MSE	PTQ-MSE	Nonzero Count	Params
Poly	No	STLSQ	0.0666	0.2493	13	thresh = 0.77, α = 1.67e-3
Poly	No	ConstrainedSR3	0.0226	0.6934	56	max = 10.0, α = 1.29e-4
Poly	Yes	STLSQ	0.0666	0.2493	13	thresh = 0.077, α = 1.67e-3
Poly	Yes	ConstrainedSR3	0.0226	0.7046	56	max = 10.0, α = 1.29e-4
HW	No	STLSQ	0.2124	0.3600	7	thresh = 0.013, α = 1.0e5
HW	No	ConstrainedSR3	0.1946	0.1951	109	max = 10.0, α = 1.29e-4
HW	Yes	STLSQ	0.2124	0.3600	7	thresh = 0.013, α = 1.0e5
HW	Yes	ConstrainedSR3	0.1946	0.1978	109	max = 10.0, α = 1.0e-5

To provide a concrete example of the discovered controllers, the following equations show the symbolic policy for the standard polynomial library with the STLSQ optimizer, before and after quantization. This variant was chosen as it is the sparsest and therefore the easiest on the eyes. The complete set of equations for all variants can be found in Appendix B.

Polynomial library, regular STLSQ (Before Quantization)

$$\begin{aligned} T = 0.11903 &- 0.99504\,y - 1.20353\,\dot{\theta} - 2.70196\,xy - 2.97954\,x\dot{\theta} \\ &+ 0.65041\,x^2y - 1.38138\,x^2\dot{\theta} + 0.13420\,y^2\dot{\theta} - 0.31545\,\dot{\theta}^3 \\ &+ 0.47202\,x^3y - 0.21937\,x^3\dot{\theta} + 0.25497\,xy^3 - 0.08639\,x\dot{\theta}^3 \end{aligned}$$

Polynomial library, regular STLSQ (After Quantization)

$$
\begin{aligned}
T = 0.12500 &- 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\
&+ 0.62500\, x^2 y - 1.50000\, x^2 \dot{\theta} + 0.14062\, y^2 \dot{\theta} - 0.31250\, \dot{\theta}^3 \\
&+ 0.46875\, x^3 y - 0.21875\, x^3 \dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3
\end{aligned}
$$

Polynomial library, QAT STLSQ

$$
\begin{aligned}
T = 0.12500 &- 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\
&+ 0.62500\, x^2 y - 1.50000\, x^2 \dot{\theta} + 0.14062\, y^2 \dot{\theta} - 0.31250\, \dot{\theta}^3 \\
&+ 0.46875\, x^3 y - 0.21937\, x^3 \dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3
\end{aligned}
$$

When comparing the solutions (found in Appendix B) obtained using the same optimizer and function library, it can be concluded that Quantization-Aware Training has no significant impact on the results. The pre-quantization MSE remains unchanged, and the final symbolic equations are identical. This is likely due to the relatively high bitwidth used during quantization—specifically, 4 integer bits and 6 fractional bits—which introduces very low quantization error.

4.4 Final Results

The final stage of the implementation involves a rigorous evaluation of the distilled "student" policies against the original RL "teacher" policy. To ensure a fair and robust comparison, each of the eight SINDy variants, along with the RL baseline, was evaluated over 10 separate environment rollouts with different initial conditions. For this evaluation, all SINDy policies were quantized post-training. The performance was assessed using two key metrics: the average episodic reward, which measures control effectiveness, and the mean squared error between the state trajectories of the student and teacher policies, which measures behavioral similarity.
The aggregated results of this evaluation are presented in Table 6.

Table 6: Performance comparison of RL and SINDy policies over 10 environment rollouts.

Policy	**Avg. Reward**	**Avg. MSE (States)**
RL (PPO)	-168.62 ± 109.42	–
poly_regular_STLSQ	-212.42 ± 143.47	0.5209 ± 0.5575
poly_regular_ConstrainedSR3	-182.40 ± 132.53	0.2546 ± 0.4445
poly_hardware_STLSQ	-212.42 ± 143.47	0.5209 ± 0.5575
poly_hardware_ConstrainedSR3	-182.40 ± 132.53	0.2546 ± 0.4445
hw_regular_STLSQ	-1071.93 ± 231.81	3.7451 ± 0.3648
hw_regular_ConstrainedSR3	-1117.74 ± 112.84	3.9603 ± 0.2625
hw_hardware_STLSQ	-1071.93 ± 231.81	3.7451 ± 0.3648
hw_hardware_ConstrainedSR3	-1117.74 ± 112.84	3.9603 ± 0.2625

The results clearly indicate a performance divide between the two function libraries. The policies generated using the standard polynomial library achieve average rewards that are

comparable to the RL teacher, with the 'poly-regular-ConstrainedSR3' variant performing particularly well. This demonstrates that the distillation process is capable of capturing the essential control strategy of the teacher policy. Conversely, all variants using the custom hardware (HW) library exhibit a significant degradation in performance, with much lower rewards and higher state trajectory error. This suggests that while the HW library is designed for implementation efficiency, its restricted functional form may not be expressive enough to accurately approximate the complex, nonlinear behavior of the teacher policy for this environment.
To provide a more granular, qualitative understanding of these results, Figure 9 visualizes the state and action trajectories for a single rollout, comparing the RL teacher against the best-performing polynomial variant ('poly_regular_STLSQ').

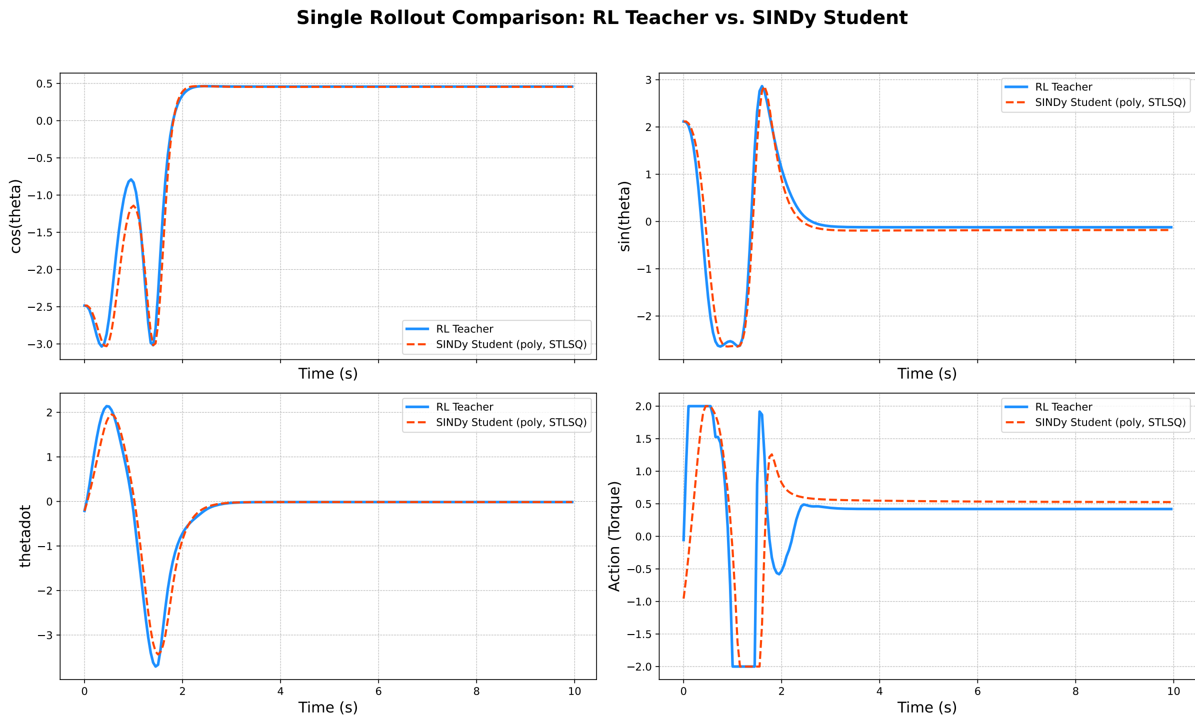

Figure 9: A comparison of state and action trajectories over time between the "teacher" (RL) and a "student" (SINDy) policy for a single rollout.

The trajectory plot confirms the quantitative findings. The SINDy "student" policy successfully mimics the overall behavior of the RL "teacher," closely tracking its state evolution and control actions. This visual evidence further validates that the distillation from a complex neural network to a sparse, symbolic equation is not only feasible but can also preserve the essential characteristics of the original high-performance controller.

5 Recommendations and Future Work

This project successfully demonstrated a two-stage pipeline that distills deep reinforcement learning policies into hardware-efficient symbolic controllers using SINDy. However, several areas present opportunities for improvement and future exploration.

5.1 Reinforcement Learning

- **Expand to more complex environments:** While the `pendulum-v1` environment provided a useful testbed, applying the pipeline to higher-dimensional tasks (e.g., quadrotors or flapping-wing MAVs) would validate its scalability and practical utility.
- **Improve evaluation robustness:** Instead of relying on a single validation split, using K-fold cross-validation allows for a more reliable assessment of model performance. This approach reduces variance in evaluation and helps detect overfitting, particularly when training data is limited.

5.2 SINDy and Policy Distillation

- **Function library design:** The quality of the SINDy approximation is heavily influenced by the expressiveness and numerical stability of the function library. Future work could focus on automating the generation or selection of library functions using data-driven methods or symbolic reasoning.
- **Adaptive regularization:** The sparsity and accuracy of the SINDy model depend on hyperparameters such as regularization weights. Introducing adaptive or learnable regularization could improve results without extensive manual tuning.

5.3 Hardware-Oriented Optimizations

- **Bitwidth:** The effect of varying quantization bitwidths on the control policy's performance should be systematically evaluated. This includes identifying the failure point where the bitwidth is too low to maintain stable and effective control
- **Custom loss shaping:** Different functions incur different hardware costs, whether in terms of power, area, or latency. By adding a term to the optimizer's loss function that penalizes high-cost functions, the optimizer can be guided to favor solutions that are more efficient with respect to these metrics.

5.4 Pipeline Extension and Real-World Deployment

- **Drone integration:** As a natural next step, the distilled policies should be tested on a simulated or physical DelFly system. This will help evaluate robustness under real-world dynamics and disturbances.
- **Toolchain automation:** The current process of generating, evaluating, and exporting SINDy models involves several manual steps. Automating this pipeline would improve usability and make it accessible to a broader audience.

5.5 Reconsideration of Requirements and Scope

- **Performance metrics:** In future projects, evaluation should go beyond episode rewards and include interpretability, latency, power consumption, and failure rates under perturbations.

6 Conclusion

This thesis successfully developed, implemented, and validated an end-to-end pipeline for converting a complex, neural network-based controller into a sparse, hardware-efficient symbolic policy. By first training a high-performance "teacher" policy with Reinforcement Learning and then distilling its knowledge using the SINDy framework, we have demonstrated a viable pathway for creating controllers suitable for resource-constrained platforms like the DelFly.

The primary contribution of this work lies in the development and evaluation of novel hardware-aware optimizations integrated directly into the SINDy distillation process. Our main innovations were:

1. **Sparse Bit Quantization (SBQ):** We introduced and implemented a new quantization scheme, SBQ, which represents coefficients as combinations of powers of two. This method is designed to replace computationally expensive multiplications with highly efficient bit-shift and addition operations, making the resulting controller inherently more compatible with hardware.

2. **Analysis of Quantization-Aware Training (QAT):** We systematically investigated the impact of performing quantization during the optimization process (QAT) versus after (post-training quantization). Our results showed that for the bit-width tested, QAT provided no significant performance benefit, a valuable finding for simplifying future hardware-oriented optimization pipelines.

3. **A Hardware-Efficient Function Library:** We designed and tested a custom SINDy library whose functions map directly to the structure of hardware DSP blocks, and analyzed the resulting trade-off between hardware efficiency and model expressiveness.

Our results on the 'pendulum-v1' proof-of-concept showed that SINDy policies using a standard polynomial library could accurately approximate the performance of the RL teacher. While the policies generated with our hardware-efficient library were sparser, they could not fully capture the complexity of the control task, highlighting a clear trade-off between hardware-compatibility and policy fidelity.

In conclusion, this work not only provides a successful proof-of-concept for the RL-SINDy pipeline but also contributes novel, hardware-oriented techniques that are essential for bridging the gap between powerful modern control algorithms and their practical deployment on real-world robotic systems.

A Terminology

Reinforcement learning and SINDy involve many technical terms that may be unclear to some readers. This appendix offers a glossary of key terms and their explanations for clarity.

Reinforcement Learning (RL): A machine learning paradigm where an agent learns to make decisions by interacting with an environment, receiving feedback in the form of rewards.

Markov Decision Process (MDP): A mathematical framework used to describe the RL problem, consisting of states, actions, transition probabilities, rewards, and a discount factor.

Gradient step: A gradient step is a single update to a model's parameters in the direction that reduces the loss, based on the gradient of the loss function.

Policy (π): A function or strategy that defines the agent's behavior by mapping states to actions.

Actor-Critic Algorithms: A class of RL methods that use separate models for the policy (actor) and value estimation (critic), such as PPO, SAC, and TD3.

Proximal Policy Optimization (PPO): An on-policy RL algorithm that uses a clipped objective to ensure stable and cautious policy updates.

Soft Actor-Critic (SAC): An off-policy RL algorithm that incorporates entropy maximization to encourage exploration and robustness.

Twin Delayed Deep Deterministic Policy Gradient (TD3): An off-policy RL algorithm that mitigates overestimation bias in value functions using twin critics and delayed updates.

SINDy (Sparse Identification of Nonlinear Dynamics): A data-driven method for identifying sparse, interpretable equations that describe dynamical systems from data.

SINDy with Control (SINDy-C): An extension of SINDy that includes control inputs in the learned models, enabling use in control systems.

PySINDy: A Python package that implements the SINDy algorithm, supporting various optimizers, function libraries, and extensions.

Function Library: A set of candidate mathematical functions (e.g., polynomials, trigonometric functions) used by SINDy to construct the symbolic model.

Sparse Optimizers: Optimization techniques used in SINDy to enforce sparsity in the model. Examples include STLSQ, Lasso, and SR3.

STLSQ (Sequentially Thresholded Least Squares): An iterative method that alternates between solving least-squares and zeroing out small coefficients to enforce sparsity.

SR3 (Sparse Relaxed Regularized Regression): An optimization method that decouples data fitting and sparsity using an auxiliary variable, offering better noise resilience.

Constrained SR3: A variant of SR3 that uses strict constraints instead of soft penalties to enforce sparsity.

Policy Distillation: The process of approximating a complex neural network policy with a simpler model, such as a SINDy-generated symbolic expression.

Quantization: The process of mapping continuous values (such as coefficients) to a finite set of discrete values suitable for hardware implementation.

Fixed-Point Representation: A numerical representation where a fixed number of bits is used for the integer and fractional parts of a number.

Sparse Bit Quantization (SBQ): A quantization technique that represents coefficients using combinations of power-of-two terms, enabling efficient implementation via bit-shifting.

Quantization-Aware Training (QAT): A training method where quantization is applied during optimization to produce models that are inherently compatible with hardware constraints.

Hardware-Efficient Function Library: A set of pre-defined expressions optimized for mapping directly to FPGA DSP blocks, promoting efficient hardware execution.

Mean Squared Error (MSE): A common metric used to quantify the difference between two functions or datasets, such as the student and teacher policies.

FPGA (Field-Programmable Gate Array): A hardware platform composed of reconfigurable logic blocks, used here as the target platform for implementing the final controller.

B Extra results

B.1 Versions and seeds

In this section, the versions of Python and the libraries used in this thesis are listed, along with the random seeds employed for reproducibility.

Versions : These are the versions of Python and the relevant libraries used for this thesis.

Package	**Version**
Python	3.12.2
dill	0.3.9
gymnasium	1.0.0
imageio	2.36.1
imageio-ffmpeg	0.6.0
joblib	1.4.0
matplotlib	3.8.3
matplotlib-inline	0.1.6
numpy	1.26.4
pandas	2.2.1
pysindy	1.7.5
scikit-learn	1.4.2
sympy	1.13.1
torch	2.6.0
torchsummary	1.5.1
torchvision	0.21.0

Table 7: Python and library versions used for this thesis.

Seeds : Whenever a single experiment was done seed "0" was used. Whenever multiple trials where tested the seed used was the respective trial number.

B.2 Additional SINDy equations for the pendulum

Polynomial library regular STLSQ after Quantization

$$\begin{aligned} T = {} & 0.12500 - 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\ & + 0.62500\, x^2 y - 1.50000\, x^2\dot{\theta} + 0.14062\, y^2\dot{\theta} - 0.31250\, \dot{\theta}^3 \\ & + 0.46875\, x^3 y - 0.21875\, x^3\dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3 \end{aligned}$$

Polynomial library QAT STLSQ

$$\begin{aligned} T = {} & 0.12500 - 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\ & + 0.62500\, x^2 y - 1.50000\, x^2\dot{\theta} + 0.14062\, y^2\dot{\theta} - 0.31250\, \dot{\theta}^3 \\ & + 0.46875\, x^3 y - 0.21875\, x^3\dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3 \end{aligned}$$

Polynomial library constrained SR3 after Quantization

$$\begin{aligned} T = {} & 0.12500 - 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\ & + 0.62500\, x^2 y - 1.50000\, x^2\dot{\theta} + 0.14062\, y^2\dot{\theta} - 0.31250\, \dot{\theta}^3 \\ & + 0.46875\, x^3 y - 0.21875\, x^3\dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3 \end{aligned}$$

Polynomial library QAT constrained SR3

$$\begin{aligned} T = {} & 0.12500 - 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\ & + 0.62500\, x^2 y - 1.50000\, x^2\dot{\theta} + 0.14062\, y^2\dot{\theta} - 0.31250\, \dot{\theta}^3 \\ & + 0.46875\, x^3 y - 0.21875\, x^3\dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3 \end{aligned}$$

Hardware library regular STLSQ after Quantization

$$\begin{aligned} T = {} & 0.12500 - 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\ & + 0.62500\, x^2 y - 1.50000\, x^2\dot{\theta} + 0.14062\, y^2\dot{\theta} - 0.31250\, \dot{\theta}^3 \\ & + 0.46875\, x^3 y - 0.21875\, x^3\dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3 \end{aligned}$$

Hardware library QAT STLSQ

$$\begin{aligned} T = {} & 0.12500 - 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\ & + 0.62500\, x^2 y - 1.50000\, x^2\dot{\theta} + 0.14062\, y^2\dot{\theta} - 0.31250\, \dot{\theta}^3 \\ & + 0.46875\, x^3 y - 0.21875\, x^3\dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3 \end{aligned}$$

Hardware library constrained SR3 after Quantization

$$\begin{aligned} T = {} & 0.12500 - 1.00000\, y - 1.25000\, \dot{\theta} - 2.50000\, xy - 3.00000\, x\dot{\theta} \\ & + 0.62500\, x^2 y - 1.50000\, x^2\dot{\theta} + 0.14062\, y^2\dot{\theta} - 0.31250\, \dot{\theta}^3 \\ & + 0.46875\, x^3 y - 0.21875\, x^3\dot{\theta} + 0.25000\, xy^3 - 0.09375\, x\dot{\theta}^3 \end{aligned}$$

Hardware library QAT constrained SR3

$$
\begin{aligned}
T = {} & 0.12500 - 1.00000\,y - 1.25000\,\dot{\theta} - 2.50000\,xy - 3.00000\,x\dot{\theta} \\
& + 0.62500\,x^2y - 1.50000\,x^2\dot{\theta} + 0.14062\,y^2\dot{\theta} - 0.31250\,\dot{\theta}^3 \\
& + 0.46875\,x^3y - 0.21875\,x^3\dot{\theta} + 0.25000\,xy^3 - 0.09375\,x\dot{\theta}^3
\end{aligned}
$$

B.3 MountainCarContinuous-v0 results

B.3.1 Environment Description

The ‘MountainCarContinuous-v0‘ environment features an underpowered car positioned between two hills. The goal is to drive the car up the right hill to reach the goal flag. Since the car’s engine is not strong enough to drive up the hill directly, the agent must learn to build momentum by driving back and forth between the hills.

- **Observation Space:** The state is described by two variables: the car’s position (from -1.2 to 0.6) and its velocity (from -0.07 to 0.07).
- **Action Space:** The agent can apply a continuous force value ranging from -1.0 to +1.0.
- **Reward Function:** The default reward is +100 for reaching the goal, with a small penalty based on the square of the action taken. This reward structure is sparse, which can make learning difficult.

Reward Shaping To facilitate more effective learning, the environment’s sparse reward was augmented using a custom reward wrapper. This technique, known as reward shaping, provides the agent with more frequent feedback. The wrapper adds a dense reward signal based on the car’s velocity, encouraging it to build momentum. The specific implementation is as follows:

```
class MountainCarContinuousRewardWrapper(gym.RewardWrapper):
    def __init__(self, env):
        super().__init__(env)

    def reward(self, reward):
        pos, vel = self.unwrapped.state
        # Encourage velocity, provide bonus for reaching goal
        shaped_reward = reward + 5 * abs(vel) - 1
        if pos >= 0.45:
            shaped_reward += 10
        return shaped_reward
```

Where the most important part is the added reward for momentum (abs(vel)).

B.3.2 Results

The learning curves for the best-performing PPO, SAC, and TD3 models identified during the grid search for the 'MountainCarContinuous-v0' environment are shown in Figure 11.

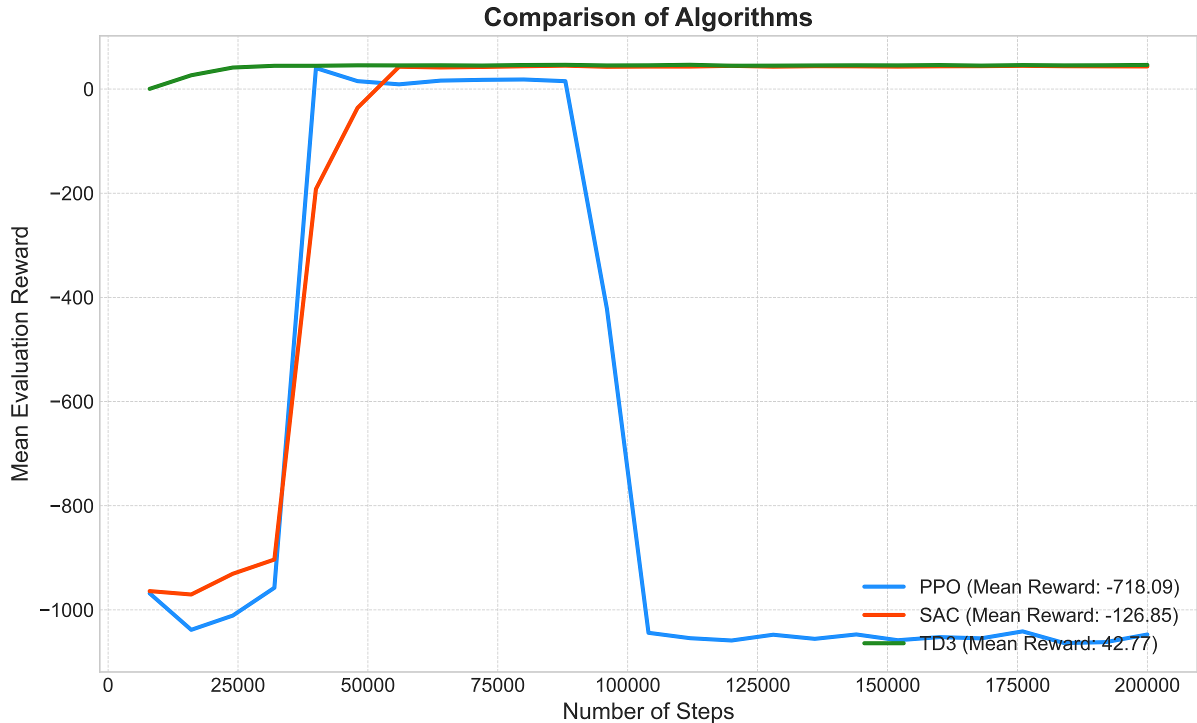

Figure 10: A comparison of the learning curves for the best models on the MountainCarContinuous-v0 environment.

The gridseach was conducted on the same combination of parameters as the pendulum environment. Notably, the PPO agent's performance collapsed after an initial period of successful learning, a common symptom of an unstable learning rate that was likely too high for sustained convergence. The candidate policies were then evaluated over 10 episodes to determine the final teacher policy. The results are presented in Table 8.

Table 8: Mean episodic reward and standard deviation for each algorithm on `MountainCarContinuous-v0`.

Algorithm	Mean Reward	Standard Deviation
PPO	-1053.19	15.62
SAC	43.54	2.05
TD3	46.03	1.94

TD3 was chosen as the best teacher policy due to the average episode length being smaller than that of SACs, indicating a more direct control strategy. The final hyperparameters found for TD3 are shown in table 8.

Table 9: Final hyperparameters for the best-performing TD3 model on `MountainCarContinuous-v0`.

Hyperparameter	**Value**
Learning Rate	0.0001
Buffer Size	500,000
Tau	0.01
Batch Size	512
Gamma (γ)	0.99
Gradient Steps	1

SINDy equations MountainCarContinuous:
Here the different SINDy variants will be presented.

poly_regular_STLSQ:

Pre-Quantized Equation

$$\begin{aligned}T = & -0.43578 - 0.30835\text{pos} + 1.40261\text{vel} + 2.00538\text{pos}^2 - 1.16853\text{pos}\cdot\text{vel} + 0.42345\text{vel}^2 \\ & + 0.19729\text{pos}^3 - 2.27286\text{pos}^2\cdot\text{vel} + 1.94063\text{pos}\cdot\text{vel}^2 - 0.75070\text{vel}^3 - 0.60009\text{pos}^4 \\ & + 0.29019\text{pos}^3\cdot\text{vel} + 0.71468\text{pos}^2\cdot\text{vel}^2 - 0.81275\text{pos}\cdot\text{vel}^3 + 0.15673\text{vel}^4 + 0.07786\text{pos}^5 \\ & + 0.29725\text{pos}^4\cdot\text{vel} - 0.32979\text{pos}^3\cdot\text{vel}^2 + 0.09446\text{pos}\cdot\text{vel}^4 + 0.01255\text{vel}^5\end{aligned}$$

Quantized Equation

$$\begin{aligned}T = & -0.43750 - 0.31250\text{pos} + 1.50000\text{vel} + 2.00000\text{pos}^2 - 1.12500\text{pos}\cdot\text{vel} + 0.43750\text{vel}^2 \\ & + 0.18750\text{pos}^3 - 2.25000\text{pos}^2\cdot\text{vel} + 1.93750\text{pos}\cdot\text{vel}^2 - 0.75000\text{vel}^3 - 0.62500\text{pos}^4 \\ & + 0.28125\text{pos}^3\cdot\text{vel} + 0.75000\text{pos}^2\cdot\text{vel}^2 - 0.87500\text{pos}\cdot\text{vel}^3 + 0.15625\text{vel}^4 + 0.07812\text{pos}^5 \\ & + 0.31250\text{pos}^4\cdot\text{vel} - 0.31250\text{pos}^3\cdot\text{vel}^2 + 0.09375\text{pos}\cdot\text{vel}^4 + 0.01562\text{vel}^5\end{aligned}$$

Variant: poly_regular_ConstrainedSR3

Pre-Quantized Equation

$$\begin{aligned}T = & -0.43544 - 0.31901\text{pos} + 1.41663\text{vel} + 2.01921\text{pos}^2 - 1.21153\text{pos}\cdot\text{vel} + 0.44458\text{vel}^2 \\ & + 0.21291\text{pos}^3 - 2.30219\text{pos}^2\cdot\text{vel} + 1.96816\text{pos}\cdot\text{vel}^2 - 0.76814\text{vel}^3 - 0.61615\text{pos}^4 \\ & + 0.36802\text{pos}^3\cdot\text{vel} + 0.59851\text{pos}^2\cdot\text{vel}^2 - 0.73594\text{pos}\cdot\text{vel}^3 + 0.13791\text{vel}^4 + 0.06697\text{pos}^5 \\ & + 0.34363\text{pos}^4\cdot\text{vel} - 0.42364\text{pos}^3\cdot\text{vel}^2 + 0.09319\text{pos}^2\cdot\text{vel}^3 + 0.04945\text{pos}\cdot\text{vel}^4 + 0.02315\text{vel}^5\end{aligned}$$

Quantized Equation

$$
\begin{aligned}
T = & -0.43750 - 0.31250\text{pos} + 1.50000\text{vel} + 2.01562\text{pos}^2 - 1.25000\text{pos}\cdot\text{vel} + 0.43750\text{vel}^2 \\
& + 0.21875\text{pos}^3 - 2.25000\text{pos}^2\cdot\text{vel} + 1.96875\text{pos}\cdot\text{vel}^2 - 0.75000\text{vel}^3 - 0.62500\text{pos}^4 \\
& + 0.37500\text{pos}^3\cdot\text{vel} + 0.62500\text{pos}^2\cdot\text{vel}^2 - 0.75000\text{pos}\cdot\text{vel}^3 + 0.14062\text{vel}^4 + 0.06250\text{pos}^5 \\
& + 0.31250\text{pos}^4\cdot\text{vel} - 0.43750\text{pos}^3\cdot\text{vel}^2 + 0.09375\text{pos}^2\cdot\text{vel}^3 + 0.04688\text{pos}\cdot\text{vel}^4 + 0.01562\text{vel}^5
\end{aligned}
$$

Variant: poly_hardware_STLSQ

Quantized Equation

$$
\begin{aligned}
T = & -0.43750 - 0.31250\text{pos} + 1.50000\text{vel} + 2.00000\text{pos}^2 - 1.12500\text{pos}\cdot\text{vel} + 0.43750\text{vel}^2 \\
& + 0.18750\text{pos}^3 - 2.25000\text{pos}^2\cdot\text{vel} + 1.93750\text{pos}\cdot\text{vel}^2 - 0.75000\text{vel}^3 - 0.62500\text{pos}^4 \\
& + 0.28125\text{pos}^3\cdot\text{vel} + 0.75000\text{pos}^2\cdot\text{vel}^2 - 0.87500\text{pos}\cdot\text{vel}^3 + 0.15625\text{vel}^4 + 0.07812\text{pos}^5 \\
& + 0.31250\text{pos}^4\cdot\text{vel} - 0.31250\text{pos}^3\cdot\text{vel}^2 + 0.09375\text{pos}\cdot\text{vel}^4 + 0.01562\text{vel}^5
\end{aligned}
$$

Variant: poly_hardware_ConstrainedSR3

Quantized Equation

$$
\begin{aligned}
T = & -0.43750 - 0.31250\text{pos} + 1.50000\text{vel} + 2.01562\text{pos}^2 - 1.25000\text{pos}\cdot\text{vel} + 0.43750\text{vel}^2 \\
& + 0.21875\text{pos}^3 - 2.25000\text{pos}^2\cdot\text{vel} + 1.96875\text{pos}\cdot\text{vel}^2 - 0.75000\text{vel}^3 - 0.62500\text{pos}^4 \\
& + 0.37500\text{pos}^3\cdot\text{vel} + 0.62500\text{pos}^2\cdot\text{vel}^2 - 0.75000\text{pos}\cdot\text{vel}^3 + 0.14062\text{vel}^4 + 0.06250\text{pos}^5 \\
& + 0.31250\text{pos}^4\cdot\text{vel} - 0.43750\text{pos}^3\cdot\text{vel}^2 + 0.09375\text{pos}^2\cdot\text{vel}^3 + 0.04688\text{pos}\cdot\text{vel}^4 + 0.01562\text{vel}^5
\end{aligned}
$$

Variant: hw_regular_STLSQ

Pre-Quantized Equation

$$
\begin{aligned}
T = & -0.03381\text{pos} + 0.03107\text{vel} + 0.01770(2\text{pos}^2 + \text{pos}) \\
& + 0.08258(2\text{pos}^2 + \text{vel}) + 0.01169(2\text{pos}^2 + 1) + 0.02010(2\text{pos}\cdot\text{vel} + \text{pos}) \\
& + 0.08498(2\text{pos}\cdot\text{vel} + \text{vel}) + 0.01409(2\text{pos}\cdot\text{vel} + 1) \\
& + 0.08378(\text{pos}^2 + \text{pos}\cdot\text{vel} + \text{vel}) + 0.01289(\text{pos}^2 + \text{pos}\cdot\text{vel} + 1) \\
& - 0.01630(\text{pos}\cdot\text{vel} + \text{pos} + \text{vel}^2) + 0.04858(\text{pos}\cdot\text{vel} + \text{vel}^2 + \text{vel}) \\
& - 0.02231(\text{pos}\cdot\text{vel} + \text{vel}^2 + 1) + 0.02302(\text{pos}^2 + \text{pos} + \text{vel}) \\
& - 0.04788(\text{pos}^2 + \text{pos} + 1) + 0.02422(\text{pos}\cdot\text{vel} + \text{pos} + \text{vel}) \\
& + 0.08910(\text{pos}\cdot\text{vel} + 2\text{vel}) + 0.01821(\text{pos}\cdot\text{vel} + \text{vel} + 1) \\
& - 0.05270(\text{pos} + 2\text{vel}^2) + 0.01218(2\text{vel}^2 + \text{vel}) - 0.05871(2\text{vel}^2 + 1) \\
& - 0.04067(\text{pos}\cdot\text{vel} + 2\text{pos}) - 0.04667(\text{pos}\cdot\text{vel} + \text{pos} + 1) \\
& - 0.01218(\text{pos} + \text{vel}^2 + \text{vel}) + 0.05270(\text{vel}^2 + 2\text{vel}) \\
& - 0.01819(\text{vel}^2 + \text{vel} + 1) - 0.10143(3\text{pos}) - 0.03655(2\text{pos} + \text{vel}) \\
& - 0.10744(2\text{pos} + 1) + 0.02833(\text{pos} + 2\text{vel}) + 0.09321(3\text{vel}) + 0.02232(2\text{vel} + 1)
\end{aligned}
$$

Quantized Equation

$$\begin{aligned}T = & -0.03125\text{pos} + 0.03125\text{vel} + 0.01562(2\text{pos}^2 + \text{pos}) \\ & + 0.07812(2\text{pos}^2 + \text{vel}) + 0.01562(2\text{pos}^2 + 1) + 0.01562(2\text{pos} \cdot \text{vel} + \text{pos}) \\ & + 0.07812(2\text{pos} \cdot \text{vel} + \text{vel}) + 0.01562(2\text{pos} \cdot \text{vel} + 1) \\ & + 0.07812(\text{pos}^2 + \text{pos} \cdot \text{vel} + \text{vel}) + 0.01562(\text{pos}^2 + \text{pos} \cdot \text{vel} + 1) \\ & - 0.01562(\text{pos} \cdot \text{vel} + \text{pos} + \text{vel}^2) + 0.04688(\text{pos} \cdot \text{vel} + \text{vel}^2 + \text{vel}) \\ & - 0.01562(\text{pos} \cdot \text{vel} + \text{vel}^2 + 1) + 0.01562(\text{pos}^2 + \text{pos} + \text{vel}) \\ & - 0.04688(\text{pos}^2 + \text{pos} + 1) + 0.03125(\text{pos} \cdot \text{vel} + \text{pos} + \text{vel}) \\ & + 0.09375(\text{pos} \cdot \text{vel} + 2\text{vel}) + 0.01562(\text{pos} \cdot \text{vel} + \text{vel} + 1) \\ & - 0.04688(\text{pos} + 2\text{vel}^2) + 0.01562(2\text{vel}^2 + \text{vel}) - 0.06250(2\text{vel}^2 + 1) \\ & - 0.04688(\text{pos} \cdot \text{vel} + 2\text{pos}) - 0.04688(\text{pos} \cdot \text{vel} + \text{pos} + 1) \\ & - 0.01562(\text{pos} + \text{vel}^2 + \text{vel}) + 0.04688(\text{vel}^2 + 2\text{vel}) \\ & - 0.01562(\text{vel}^2 + \text{vel} + 1) - 0.09375(3\text{pos}) - 0.03125(2\text{pos} + \text{vel}) \\ & - 0.10938(2\text{pos} + 1) + 0.03125(\text{pos} + 2\text{vel}) + 0.09375(3\text{vel}) + 0.01562(2\text{vel} + 1)\end{aligned}$$

Variant: hw_regular_ConstrainedSR3

Pre-Quantized Equation

$$\begin{aligned}T = & \, 0.10011(2\text{pos}^2 + \text{vel}) + 0.08276(2\text{pos} \cdot \text{vel} + \text{vel}) \\ & + 0.09552(\text{pos}^2 + \text{pos} \cdot \text{vel} + \text{vel}) + 0.12812(\text{pos} \cdot \text{vel} + 2\text{vel}) \\ & - 0.03546(\text{pos} + 2\text{vel}^2) - 0.04763(2\text{vel}^2 + 1) - 0.12852(3\text{pos}) \\ & - 0.17438(2\text{pos} + 1) + 0.15720(3\text{vel})\end{aligned}$$

Quantized Equation

$$\begin{aligned}T = & \, 0.09375(2\text{pos}^2 + \text{vel}) + 0.07812(2\text{pos} \cdot \text{vel} + \text{vel}) \\ & + 0.09375(\text{pos}^2 + \text{pos} \cdot \text{vel} + \text{vel}) + 0.12500(\text{pos} \cdot \text{vel} + 2\text{vel}) \\ & - 0.03125(\text{pos} + 2\text{vel}^2) - 0.04688(2\text{vel}^2 + 1) - 0.12500(3\text{pos}) \\ & - 0.18750(2\text{pos} + 1) + 0.15625(3\text{vel})\end{aligned}$$

Variant: hw_hardware_STLSQ

Quantized Equation

$$
\begin{aligned}
T = & -0.03125\text{pos} + 0.03125\text{vel} + 0.01562(2\text{pos}^2 + \text{pos}) \\
& + 0.07812(2\text{pos}^2 + \text{vel}) + 0.01562(2\text{pos}^2 + 1) + 0.01562(2\text{pos} \cdot \text{vel} + \text{pos}) \\
& + 0.07812(2\text{pos} \cdot \text{vel} + \text{vel}) + 0.01562(2\text{pos} \cdot \text{vel} + 1) \\
& + 0.07812(\text{pos}^2 + \text{pos} \cdot \text{vel} + \text{vel}) + 0.01562(\text{pos}^2 + \text{pos} \cdot \text{vel} + 1) \\
& - 0.01562(\text{pos} \cdot \text{vel} + \text{pos} + \text{vel}^2) + 0.04688(\text{pos} \cdot \text{vel} + \text{vel}^2 + \text{vel}) \\
& - 0.01562(\text{pos} \cdot \text{vel} + \text{vel}^2 + 1) + 0.01562(\text{pos}^2 + \text{pos} + \text{vel}) \\
& - 0.04688(\text{pos}^2 + \text{pos} + 1) + 0.03125(\text{pos} \cdot \text{vel} + \text{pos} + \text{vel}) \\
& + 0.09375(\text{pos} \cdot \text{vel} + 2\text{vel}) + 0.01562(\text{pos} \cdot \text{vel} + \text{vel} + 1) \\
& - 0.04688(\text{pos} + 2\text{vel}^2) + 0.01562(2\text{vel}^2 + \text{vel}) - 0.06250(2\text{vel}^2 + 1) \\
& - 0.04688(\text{pos} \cdot \text{vel} + 2\text{pos}) - 0.04667(\text{pos} \cdot \text{vel} + \text{pos} + 1) \\
& - 0.01562(\text{pos} + \text{vel}^2 + \text{vel}) + 0.04688(\text{vel}^2 + 2\text{vel}) \\
& - 0.01562(\text{vel}^2 + \text{vel} + 1) - 0.09375(3\text{pos}) - 0.03125(2\text{pos} + \text{vel}) \\
& - 0.10938(2\text{pos} + 1) + 0.03125(\text{pos} + 2\text{vel}) + 0.09375(3\text{vel}) + 0.01562(2\text{vel} + 1)
\end{aligned}
$$

Variant: hw_hardware_ConstrainedSR3

Quantized Equation

$$
\begin{aligned}
T = & \; 0.09375(2\text{pos}^2 + \text{vel}) + 0.07812(2\text{pos} \cdot \text{vel} + \text{vel}) \\
& + 0.09375(\text{pos}^2 + \text{pos} \cdot \text{vel} + \text{vel}) + 0.12500(\text{pos} \cdot \text{vel} + 2\text{vel}) \\
& - 0.03125(\text{pos} + 2\text{vel}^2) - 0.04688(2\text{vel}^2 + 1) - 0.12500(3\text{pos}) \\
& - 0.18750(2\text{pos} + 1) + 0.15725(3\text{vel})
\end{aligned}
$$

Here is a comparison between the teacher and student policy and the states and actions over time:

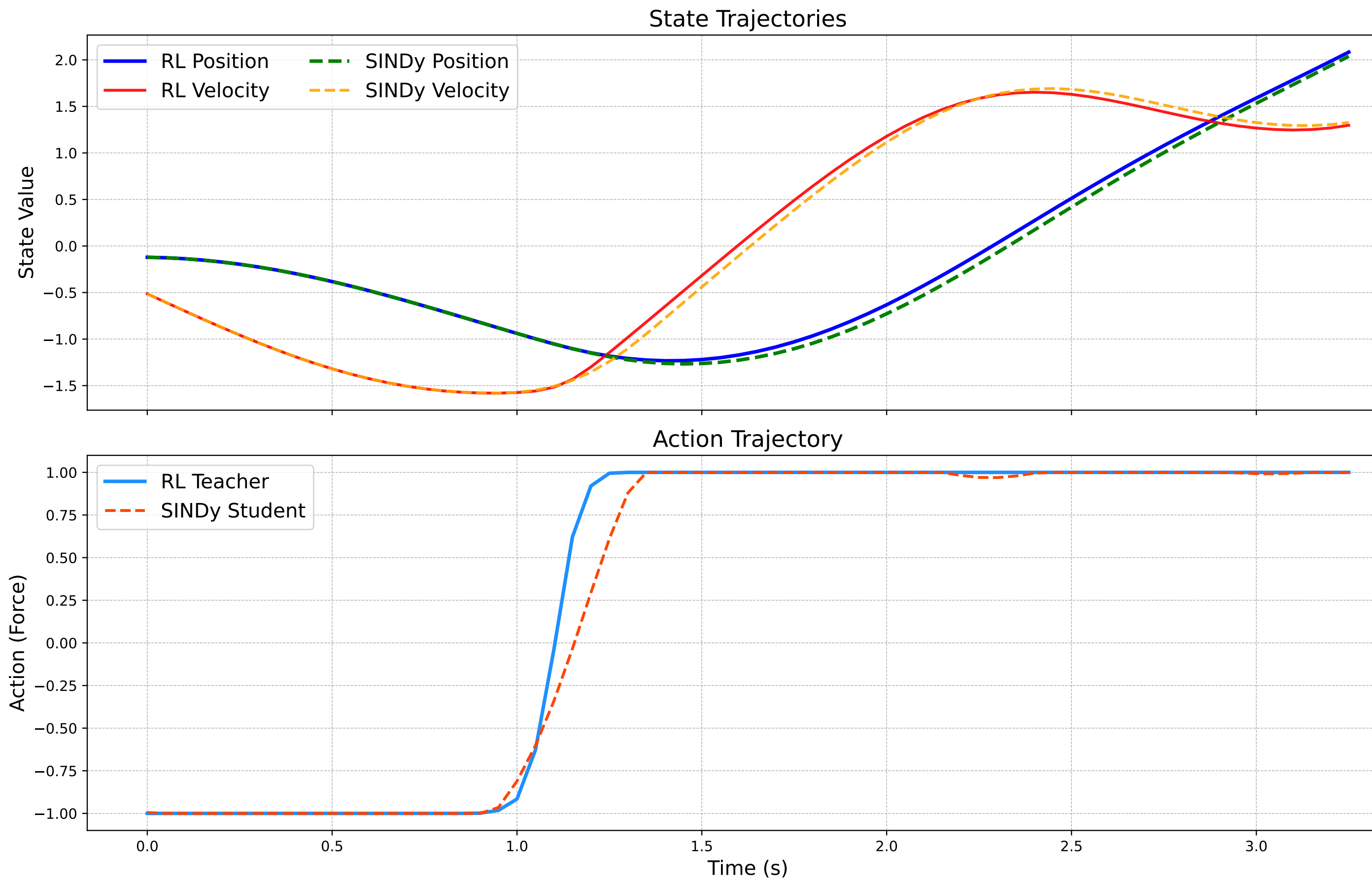

Figure 11: A comparison between the states and the action of the RL and SINDy policies for a certain trial.

What you can see is that the states and actions are followed nicely by the student policy with regards to the teacher policy.

B.4 CartPole-v1 results

B.4.1 Environment Description

'CartPole-v1' is a classic balancing problem. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The goal is to prevent the pole from falling over by applying force to the cart.

- **Observation Space:** The state is described by four variables: the cart's position, its velocity, the pole's angle, and the pole's angular velocity.
- **Action Space:** The agent can take one of two discrete actions: push the cart to the left (0) or push it to the right (1).
- **Reward Function:** The agent receives a reward of +1 for every step taken where the pole remains upright. The episode ends if the pole falls past a certain angle or the cart moves off the edge of the track.

B.4.2 Results

Since CartPole-v1 uses a discrete action space, only PPO was suitable as SAC and TD3 only support discrete action spaces. Additionally, a different gridsearchspace was searched as can be seen in table 9.

Table 10: Hyperparameter search space for the PPO algorithm grid search.

Hyperparameter	**Values Searched**
Learning Rate	5e-4, 1e-4, 5e-5
Gamma (γ)	0.98, 0.999
Batch Size	256
N_Steps	1024
Entropy Coefficient	0.005, 0.05, 0.3

The final hyperparameters found for the best-performing model are shown in table 10.

Table 11: Final hyperparameters for the best-performing PPO model identified during the grid search.

Hyperparameter	**Value**
Learning Rate	5e-05
Gamma (γ)	0.98
Batch Size	256
N_Steps	1024
Entropy Coefficient	0.05

The learning curves for the best-performing PPO model was identified during the grid search for the 'CartPole-v1' and environment is shown in Figure 13.

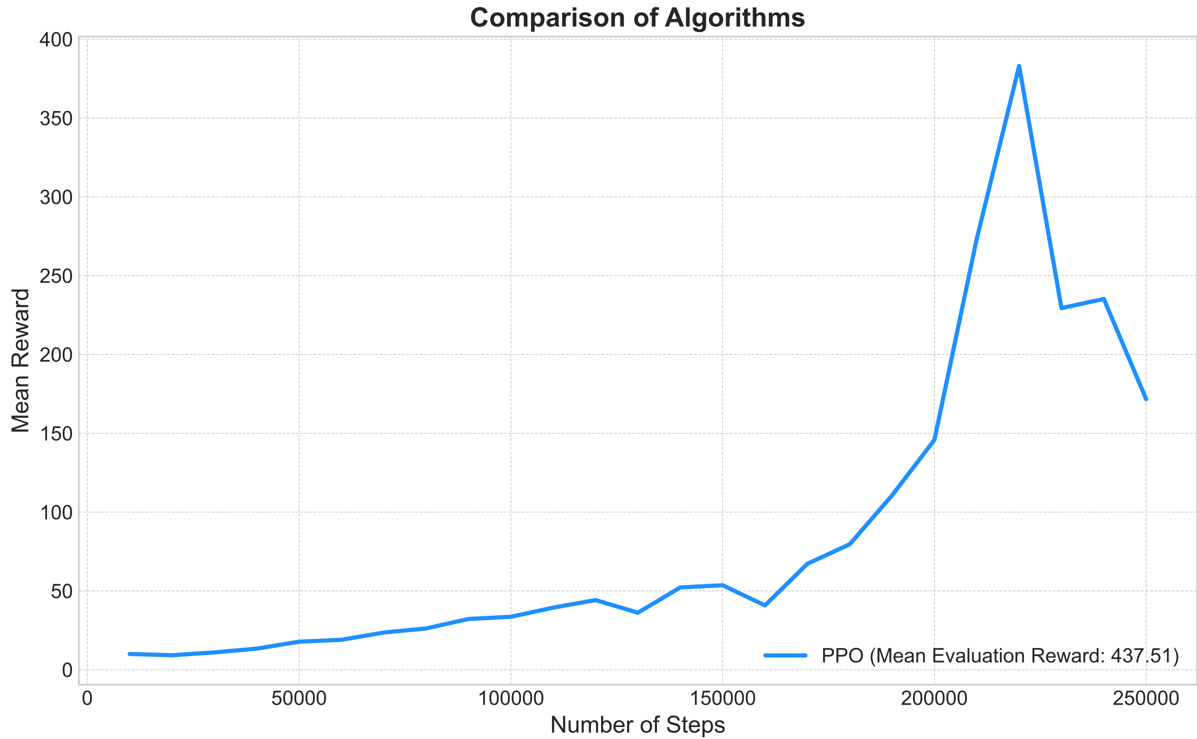

Figure 12: The learning curve for the best model on the CartPole-v1 environment.

The learning curve in Figure 11 suggests the training was still improving and had not yet converged. The final mean reward is higher than the peak of this curve, a discrepancy that likely arises from the evaluation methodology. The curve plots periodic performance snapshots, while the final reward is a more stable average calculated over a dedicated set of evaluation episodes post-training. The candidate policy was then evaluated over 10 episodes to determine the final teacher policy. The results are presented in Table 12.

Table 12: Mean episodic reward and standard deviation for each algorithm on CartPole-v1.

Algorithm	**Mean Reward**	**Standard Deviation**
PPO	500	0

Finally here are the found sindy equations :
Here is a comparison between the teacher and student policy and the states and actions over time:

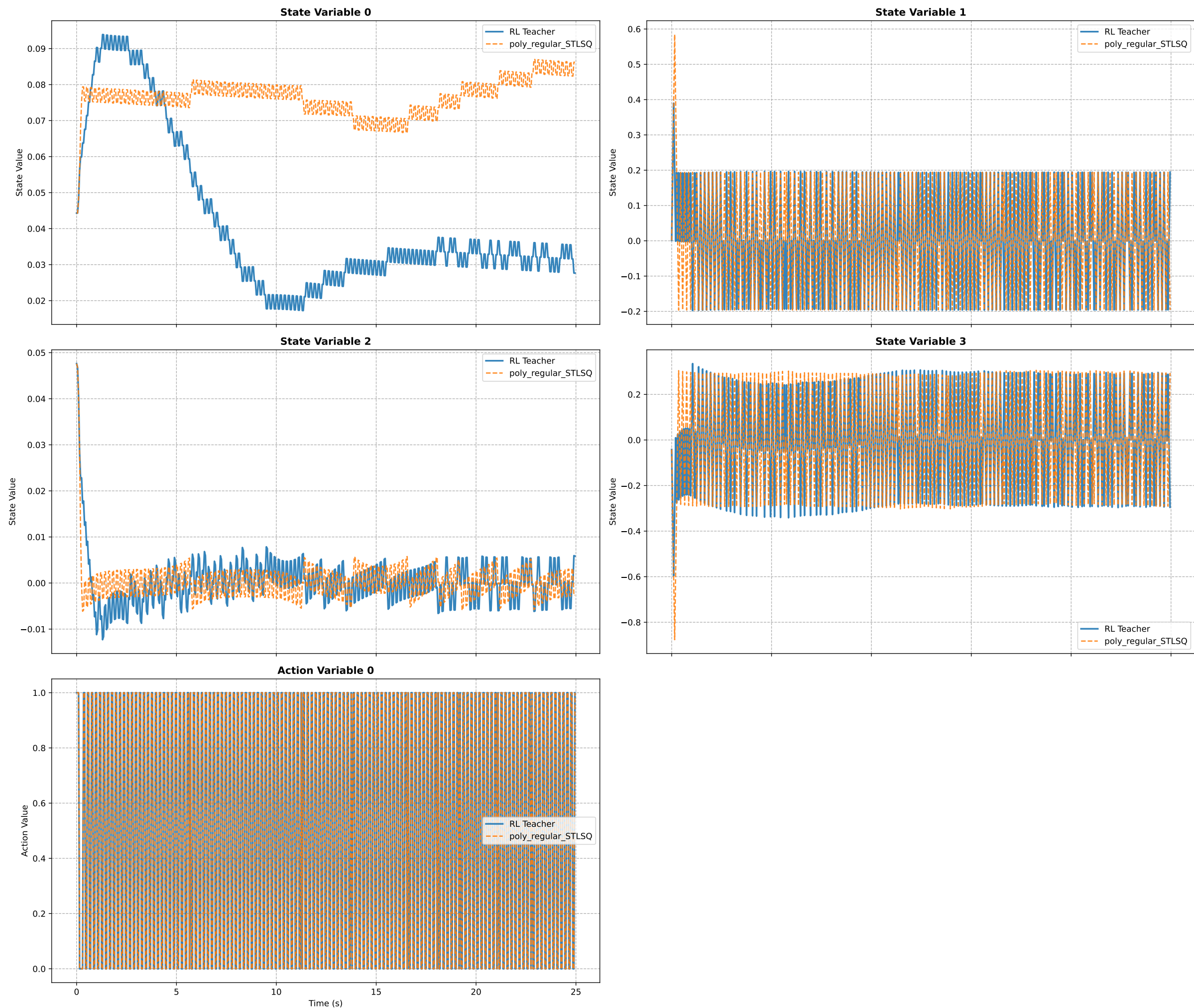

Figure 13: A comparison of the RL and SINDy policies of CartPole-v1 .

The seemingly noisy behavior observed in the state trajectory plots is actually characteristic of a successful control strategy for the CartPole environment. This oscillatory motion stems from two fundamental properties of the system:
First, the action space is discrete, meaning the agent can only apply a push to the left or to the right; it cannot remain perfectly still or apply a finely-tuned force. To keep the pole balanced, the agent must constantly make small, rapid corrections by alternating these pushes.
This leads to the constant back-and-forth oscillations seen across all four state variables: the cart position (State 0), cart velocity (State 1), pole angle (State 2), and pole angular velocity (State 3). The effectiveness of this balancing act is confirmed by observing that the average values for the cart's velocity, the pole's angle, and its angular velocity all remain close to zero, demonstrating that the agent is actively and successfully stabilizing the system around its upright setpoint.

C Delfly environment

To use RL on the flapping wing drone an environment for gymnasium must be made first. In this environment the agent will be able to take actions and receive rewards and observation. The following is needed to create an environment:

- **Action Space** which is a set of actions the agent can take.
- **Observation space** which is the set of states the agent can be in.
- **Reset function** which is called at the beginning of each episode to initialize all variables
- **Step function** which is used each iteration to determine the action the agent will take.
- **Render function** which is optional if you want to render the agent and the environment

Most environment have also a helper function which determines the next state given a state. This function is usually refered to as dynamics, since it captures the dynamics of the system. The helper function gets called in the step function.
To determine what the spaces and functions should look like for the Delfly, [26] was used. From the paper the following were obtained:

C.1 Action space

The actions that we can control are the flapping frequency and the dihedral angle. The dihedral angle is the angle of the wings closed position, which may vary between -18 and 18 degrees. The degrees will be converted to radians since most functions work with radians. The frequency can vary between 0 and 22 HZ. So the frequency f and the dihedral angle γ can take continuous values in the ranges:

$$f \in [0, 22], \quad \theta \in [-0.3142, -.3142]$$

C.2 State Space

The following states will be observerd in the environment:

- **u** which stand for the the longitudonal speed with respect to the body
- **w** which stand for the the vertical speed with respect to the body
- θ which stand for the pitch of the drone with respect to the world. This is defined as the angle between the negative y axis of the world and the negative w axis of the drone.
- $\dot{\theta}$ which stand for angular velocity of the pitch.

C.3 Reset function

In this function only the initial state is set. Since the Delfly always starts on the ground and positionds straight up so zero pitch, The initial conditions are all set to 0. which are the longitudonal velocity, vertical velocity, pitch, angular speed of the pitch.

C.4 Step function

The step function will take in a current state and control input and output the next state. To be able to do this the dynamics of the system will be used which are the equations of motion of the Delfy.

$$m\dot{u} = -m\dot{\theta}w - mgsin\theta - b_x f(u - l_z\dot{\theta} + \dot{l}_d \quad (1)$$

$$m\dot{w} = -m\dot{\theta}u - mgcos\theta - T(f) - b_z f(w - l_d\dot{\theta}) \quad (2)$$

$$I\ddot{\theta} = -b_x f l_z(u - l_z\dot{\theta} + \dot{l}_d) + b_z f l_d(w - l_d\dot{\theta}) - T(f)l_d \quad (3)$$

using these equations and current state, values for $\dot{u}, \dot{w}, \ddot{\theta}$ are found. By integrating them using the Runga-Kutta method for one time step, next states are calculated.

C.5 Render function

For Visualizing our System plots will be used. The states and control input's will be plotted to see how they evolve over time and if they behave as expected.

C.6 Results

When training, the reward the agent receives per episode appears as noise. This suggests that the agent was unable to learn an effective policy using any of the tested RL algorithms. Given that these RL algorithms have successfully learned in various other continuous control tasks, we suspect that the issue lies in the mathematical model of the DelFly. A possible solution is to simplify the model equations as much as possible. If the agent is able to learn with the simplified model, the complexity of the equations can then be gradually increased to identify the point at which learning becomes unstable.

D Project Repository

The full source code used in this project is available on GitHub. It includes the implementation of the reinforcement learning pipeline, the SINDy-based symbolic regression, custom optimizers, and all experiments discussed in this thesis.

- GitHub Repository: `https://github.com/PabloABakker/BAP`

This repository serves as a complete reference for reproducing the results and building upon the work presented in this thesis.

E Code explanation

The important scripts that can be found in the repository will be explained here.

E.1 RL

E.1.1 train.py

This script performs hyperparameter grid search for reinforcement learning algorithms (PPO, SAC, TD3) on a specified custom environment. It trains multiple configurations in parallel with different random seeds, evaluates their performance, and saves the results (including models and training stats) for analysis, with the goal of finding the best hyperparameter combination for the given environment.

E.1.2 test.py

This script evaluates trained RL policies (PPO/SAC/TD3) by testing their performance over multiple episodes and optionally generating a GIF of their behavior. It loads models and their normalization stats, calculates mean/std rewards, and saves results as JSON while supporting custom environments.

E.1.3 gen_data_for_sindy.py

This script runs a trained RL policy while recording both normalized and raw environment states along with the policy's actions, saving the data to a CSV file for analysis. It handles model loading, environment normalization, and data collection in one streamlined process.

E.2 SINDy

E.2.1 create_sindy_policies.py

This script trains SINDy models on RL data, testing different optimizers and feature libraries while optimizing hyperparameters. It evaluates model performance with and without quantization, saving the best versions for each configuration. The parallelized implementation efficiently searches parameter spaces and stores results with full metadata.

E.2.2 compare_SINDY_vs_RL.py

This script compares reinforcement learning (RL) policies against SINDy-derived control policies across multiple trials. It evaluates performance by running both policy types in the same environment, recording trajectories, rewards, and state mismatches. The implementation handles policy execution, data collection, and visualization while supporting various RL algorithms and SINDy model configurations. Results are saved as GIF animations, trajectory plots, and CSV reports for quantitative comparison. The script ensures consistent evaluation through controlled random seeds and proper environment normalization.

E.2.3 make_ready_for_dsp_flow.py

This script extracts and formats mathematical equations from SINDy models, handling both pre-quantized and quantized versions of the coefficients. It systematically processes all model variants, applies fixed-point quantization to coefficients when specified, and converts the numerical models into human-readable equation strings. The implementation includes robust variable name substitution and proper formatting with explicit multiplication signs, while preserving the original model structure. Results are saved in a structured text file that clearly distinguishes between different model variants and their quantized states.

E.3 Custom

E.3.1 quantization.py

This script holds the SBQ class which is called FixedPointVisualizer.

E.3.2 CustomDynamicsEnv_v3.py

This custom Gymnasium environment simulates a nonlinear aerodynamic system with four state variables ($u, w, theta, theta_dot$) and two control inputs. It implements complex dynamics through numerical integration, featuring adaptive reward shaping that tightens stability thresholds as training progresses. The environment provides termination conditions for safety bounds and includes debugging outputs, while maintaining full Gymnasium API compatibility for reinforcement learning integration.

E.3.3 custom_optimizers.py

The code implements two custom sparse regression optimizers (HWConstrainedSR3 and HWConstrainedSTLSQ) that extend PySINDy's optimizers with hardware-aware constraints. Both classes incorporate a quantization step that modifies coefficients to hardware-friendly values during optimization. The HWConstrainedSR3 uses a constrained optimization approach with SLSQP, while HWConstrainedSTLSQ employs iterative thresholding. Key features include feature-specific regularization weights and quantization-aware coefficient updates that maintain model sparsity while respecting hardware constraints. The optimizers preserve the standard PySINDy interface while adding hardware-specific parameters like the quantizer and feature weights.

E.3.4 custom_libs.py

which contains functions that return lists of the library functions and there corresponding name.

F Requirements

This section revisits the project requirements from both the general project level and the software subgroup perspective. Each requirement is accompanied by a short summary evaluating whether it was met during the course of the project.

F.1 General Requirements

Requirement	Achievement Status
The system, as a baseline, must use a neural-network based controller to control an agent in a simulation environment.	**Fully met.** A neural network controller was successfully trained using three different reinforcement learning algorithms (PPO, SAC, and TD3) to control agents in multiple simulation environments.
The system must use SINDy to sparsify the neural network-based controller and provide a simplified mathematical expression representing the controller.	**Fully met.** The SINDy framework was successfully used to distill the trained RL policies into sparse, symbolic equations, as shown by the results in Table 5 and the example equations provided.
The system must be able to convert the simplified controller into an FPGA design.	**Partially met.** From the software perspective, the final symbolic equations were successfully formatted for handoff to the hardware subgroup. The actual conversion to an FPGA design is the responsibility of the hardware team.
The system's efficacy must be demonstrated for the case of a pendulum simulation environment as a proof-of-concept.	**Fully met.** The entire RL-to-SINDy pipeline was successfully implemented and validated on the `pendulum-v1` environment, with detailed results presented throughout the implementation chapter.

F.2 Software Subgroup Requirements

Table 13: Software Subgroup Requirements and Achievement Status

Requirement	Achievement Status
The trained RL controller must achieve a mean episodic reward sufficient to be considered a successful solution for the given control task.	**Fully met.** The selected PPO teacher policy achieved a mean reward of -119.46 on the `pendulum-v1` environment, indicating a high-performance and successful control strategy.
The distilled SINDy policy must achieve performance (mean episode rewards) comparable to the original RL policy that it approximates.	**Partially met.** As shown in Table 6, the SINDy policies using the polynomial library achieved comparable rewards (e.g., -182.40 for ConstrainedSR3 vs. -168.62 for the RL teacher). However, the policies using the hardware-efficient library failed to achieve comparable performance.
A dedicated hardware-oriented SINDy optimizer shall be developed and evaluated, with its performance compared against baseline optimizers.	**Fully met.** A hardware-oriented optimizer incorporating Quantization-Aware Training (QAT) was developed. Its performance was evaluated and compared against the baseline (non-QAT) optimizers in Table 5.
The SINDy distillation process must produce a sparse policy, defined by a minimal number of active terms, suitable for efficient hardware implementation.	**Fully met.** The SINDy process successfully generated sparse policies. For example, the `poly_regular_STLSQ` variant produced a policy with only 13 non-zero terms, and the `hw_regular_STLSQ` variant produced one with only 7 terms.
The effect of implementing and using a hardware-efficient function library within the SINDy framework must be evaluated in terms of final policy performance (mean episode rewards).	**Fully met.** The hardware-efficient library was implemented and evaluated. Table 6 clearly shows its effect on performance, demonstrating a significant trade-off between hardware-efficiency and control fidelity compared to the polynomial library.
The learned RL and distilled SINDy policies must demonstrate robustness by successfully stabilizing the system from a wide range of initial conditions.	**Fully met.** The final evaluation was conducted over 10 trials using different random seeds for each policy. The reported mean and standard deviation in Table 6 demonstrate the policies' performance across these varied initial conditions.

References

[1] Xinyu Zhang et al. "Wing Kinematics-Based Flight Control Strategy in Insect-Inspired Flight Systems: Deep Reinforcement Learning Gives Solutions and Inspires Controller Design in Flapping MAVs". In: *Drones* 8.1 (2024), p. 27.

[2] Matěj Karásek et al. "A tailless flapping-wing robot with control authority in all three axes". In: *Science Robotics* 3.24 (2018), eaat0354.

[3] KS Holkar and Laxman M Waghmare. "An overview of model predictive control". In: *International Journal of control and automation* 3.4 (2010), pp. 47–63.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd. MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[5] Jack Foerster, Keane Y. F. Low, and Gabriele M. T. D'Eleuterio. "Learning to fly through deep reinforcement learning". In: *Science Robotics* 5.42 (2020), eaba2821.

[6] Jonas Degrave et al. "Magnetic control of tokamak plasmas through deep reinforcement learning". In: *Nature* 602.7897 (2022), pp. 414–419.

[7] Taewi Kim et al. "Wing-strain-based flight control of flapping-wing drones through reinforcement learning". In: *Nature Machine Intelligence* 6.9 (2024), pp. 992–1005. ISSN: 2522-5839. DOI: 10.1038/s42256-024-00893-9. URL: https://doi.org/10.1038/s42256-024-00893-9.

[8] Yunlong Song et al. "Reaching the limit in autonomous racing: Optimal control versus reinforcement learning". In: *Science Robotics* 8.82 (2023), eadg1462. DOI: 10.1126/scirobotics.adg1462. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.adg1462. URL: https://www.science.org/doi/abs/10.1126/scirobotics.adg1462.

[9] Vivienne Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[10] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences* 113.15 (Mar. 2016), pp. 3932–3937. ISSN: 1091-6490. DOI: 10.1073/pnas.1517384113. URL: http://dx.doi.org/10.1073/pnas.1517384113.

[11] Andrew Boutros and Vaughn Betz. "FPGA architecture: Principles and progression". In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29.

[12] Rushiv Arora, Bruno Castro da Silva, and Eliot Moss. *Model-Based Reinforcement Learning with SINDy*. 2022. arXiv: 2208.14501 [cs.LG]. URL: https://arxiv.org/abs/2208.14501.

[13] Nicholas Zolman et al. *SINDy-RL: Interpretable and Efficient Model-Based Reinforcement Learning*. 2024. arXiv: 2403.09110 [cs.LG]. URL: https://arxiv.org/abs/2403.09110.

[14] Aniket Dixit et al. "LEARNING FROM LESS: SINDY SURROGATES IN RL". In: *ICLR 2025 Workshop on World Models: Understanding, Modelling and Scaling*. 2025. URL: https://openreview.net/forum?id=vjC6H84NCS.

[15] Mark Towers et al. *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. 2024. arXiv: 2407.17032 [cs.LG]. URL: https://arxiv.org/abs/2407.17032.

[16] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[17] Tuomas Haarnoja et al. *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.

[18] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: *arXiv preprint arXiv:1802.09477* (2018).

[19] Urban Fasel et al. "SINDy with Control: A Tutorial". In: *2021 60th IEEE Conference on Decision and Control (CDC)*. 2021, pp. 16–21. DOI: 10.1109/CDC45484.2021.9683120.

[20] Alan Kaptanoglu et al. "PySINDy: A comprehensive Python package for robust sparse system identification". In: *Journal of Open Source Software* 7.69 (Jan. 2022), p. 3994. ISSN: 2475-9066. DOI: 10.21105/joss.03994. URL: http://dx.doi.org/10.21105/joss.03994.

[21] Cheng Huang et al. "Sparse Identification of Nonlinear Dynamics via an Sequential Thresholded-Iteratively Reweighted Least Squares". In: *2023 5th International Conference on Industrial Artificial Intelligence (IAI)*. 2023, pp. 1–6. DOI: 10.1109/IAI59504.2023.10327589.

[22] J Ranstam and J A Cook. "LASSO regression". In: *British Journal of Surgery* 105.10 (Aug. 2018), pp. 1348–1348. ISSN: 0007-1323. DOI: 10.1002/bjs.10895. eprint: https://academic.oup.com/bjs/article-pdf/105/10/1348/36206240/bjs10895.pdf. URL: https://doi.org/10.1002/bjs.10895.

[23] Peng Zheng et al. "A Unified Framework for Sparse Relaxed Regularized Regression: SR3". In: *IEEE Access* 7 (2019), pp. 1404–1423. DOI: 10.1109/ACCESS.2018.2886528.

[24] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[25] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[26] K M Kajak et al. "A minimal longitudinal dynamic model of a tailless flapping wing robot for control design". In: *Bioinspiration Biomimetics* 14.4 (June 2019), p. 046008. DOI: 10.1088/1748-3190/ab1e0b. URL: https://dx.doi.org/10.1088/1748-3190/ab1e0b.