Internship Report
Second Year - Engineering Cycle
Program : Autonomous Robotics

Presented by : **Salah El Din SEKAR**

Supervised by : **Thao DANG**

University : **Ensta Bretagne**

Host University : **Université Grenoble Alpes / Verimag**

November 3, 2024

Contents

List of Figures

Abstract

This report represents my internship at Verimag, Grenoble, under the supervision of Dr. Thao Dang. The project focuses on using imitation learning to train a neural network controller that integrates the advantageous properties of conventional controllers specified using Signal Temporal Logic (STL). The implementation is applied to robotic systems utilizing the Robot Operating System (ROS2).

The core of the project involves creating high-quality data for training a neural network (NN) controller by generating behavioral traces through ROS2 simulations. By high-quality data, we mean data that is both diverse and comprehensive, ensuring good generalization ability for the resulting neural network controller. These traces are then used to train the neural network using MATLAB. To ensure the accuracy and reliability of the NN controller, the Breach tool is employed for falsification, identifying and rectifying deviations from expected behaviors. This approach aims to enhance the accuracy, robustness, and safety of the neural network controllers, making them suitable for deployment in complex, real-world robotic applications.

Introduction

Artificial intelligence (AI) and data sciences are rapidly transforming our society, leading to groundbreaking advancements across various domains. These technological innovations necessitate a twofold evolution in systems design. On one hand, new theories and methodologies must be developed to accommodate and leverage these AI advancements. On the other hand, systems design can harness the power of AI and data sciences to significantly enhance the efficiency, dependability, and security of control systems.

One of the major challenges in designing the next generation of control systems lies in the inherent unpredictability of AI techniques. Unlike traditional control methods, AI lacks a formal framework to provide robust safety guarantees, making it difficult to predict and manage potential risks. Addressing this challenge is crucial for the successful integration of AI into control systems, particularly in critical applications such as robotics.

This project aims to bridge this gap by focusing on the formal design of neural network controllers for robotic systems using imitation learning. The primary objective is to create a framework that ensures both the robustness and reliability of AI-driven controllers in real-world scenarios.

The manuscript is organized as follows. In **Chapter 1**, we introduce the concept of formal imitation learning, providing an overview of imitation learning using neural networks and the formal control specifications necessary for ensuring robust performance. We also discuss the Breach tool, which is used for formal validation and falsification of the controllers. In **Chapter 2**, we present a multiple controller imitation learning strategy, describing the control requirements and the use of Signal Temporal Logic (STL) formulas to verify robustness. We also outline the implementation strategy and discuss the verification process and results. **Chapter 3** focuses on the implementation of the project using ROS2, detailing the simulation and data generation process, as well as challenges such as down-sampling and data conversion.

Chapter 1

Formal Imitation Learning

1.1 Imitation Learning

Context and Motivation

In this section, we briefly recall the general ideas of imitation learning (IL) as well as the concrete elements of IL in our framework. We will also discuss the motivations of using IL with neural networks.

Imitation learning, also known as Learning from Demonstration (LFD) [9], is a method in machine learning that involves mimicking the behavior of an expert, often a human, to achieve some behavioral goal or optimal system performance. In our framework, a number of conventional controllers are combined to play the role of the expert, the behavior of which is then imitated by a neural network.

How It Works

1. **Data Collection:**
   - The first step involves gathering data from an expert performing a task. For instance, if the task involves controlling a water tank and the expert is a PID controller, the system states, reference inputs, and the associated control actions produced by the expert are recorded in the form of time series. The expert could also be a Model Predictive Controller (MPC), where the control inputs and corresponding system states are similarly logged.
2. **Training:**
   - The collected data is then used to train a machine learning model. In our framework, the ML models are neural networks. These models learn to map the system states and reference inputs to the desired control actions by imitating the expert's behavior.
3. **Evaluation and Counter-Example Guided Retraining:**
   - After training, the model's correctness and performance are evaluated. This involves testing the neural network controller in various scenarios to ensure it can reliably replicate the expert's behavior, which in our framework is characterized by the satisfaction of formal requirements specified using Signal Temporal Logic (STL) [1], as discussed in the next section. If the model's performance is not satisfactory, further improvements and refinements are made to the training process. In our framework, these improvements involve generating additional data to correct the unsatisfactory behavior of the neural network controller.

Applications

Before continuing, let us mention some popular applications of imitation learning:

- **Autonomous Vehicles:** Train self-driving cars by learning from human drivers, capturing complex maneuvers and real-world driving behavior.

- **Robotics:** Teach robots tasks like cooking or folding clothes, which are easy for humans but hard to program.
- **Game Playing:** Train AI agents to play video and board games at a human level by learning from skilled players.
- **Healthcare:** Assist in surgical robotics by learning from expert surgeons to perform delicate operations.

1.2 Imitation Learning using Neural Networks

1.2.1 Control Scheme

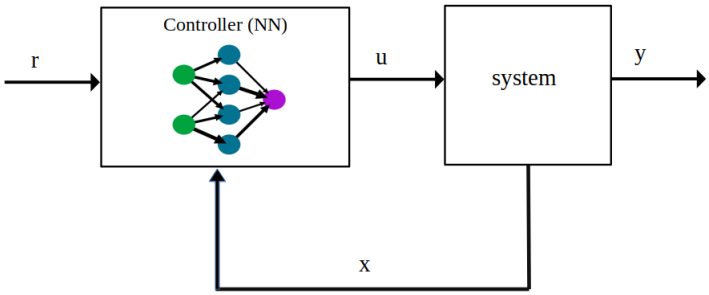

Figure 1.1: Neural Network Architecture for Imitation Learning

In this work, we explore the design of a neural network-based controller using imitation learning. This technique enables the neural network to replicate the behavior of traditional controllers. By leveraging the approximation power of neural networks, we aim to create a controller that is more efficient and easier to deploy, particularly in real-time applications.

In our architecture, the neural network learns from the expert controller's demonstrations, adapting its predictions over time to imitate the expert's behavior. The symbols r, u, y, and x represent various components in this system.

- r **(Reference Signal)**: The target behavior the controller should follow, including current and past values, like r_{prev}.
- u **(Control Input)**: The actions generated by the network based on the reference signal and state, aiming to replicate the expert controller's behavior.
- y **(Observation)**: The current measured state of the system, providing the necessary context for the network to decide the control action u.
- x **(State Representation)**: The dynamic system state, including past levels such as H, H_{prev}, used by the network to adjust its actions.

This imitation learning framework provides the neural network with temporal context and real-time observations to generate appropriate control actions u. By dynamically adjusting its predictions based on feedback from past states, the neural network achieves better performance and more robust behavior in control tasks, ultimately enhancing efficiency and ease of deployment in real-time applications.

1.2.2 Advantages of Neural Networks

Neural networks offer several key advantages when used in control systems:

- **Efficiency in Deployment:** An important motivation of using neural networks in our framework is that they are less costly to deploy than many complex controllers that requires online optimization, such as Model Predictive Control (MPC) [11]

- **Approximation Power:** Neural networks are capable of approximating complex nonlinear functions. This allows them to emulate the behavior of advanced controllers, such as MPC[2], while operating with much lower computational overhead.
- **Combining Capacities of Different Controllers** It is important to note that each conventional controller can perform well only within some operation domain. The neural network obtained by imitation learning can perform in all operation domains and additionally can achieve at the same time the properties which may be conflicting (such as fast stabilization and absence of overshoot) and cannot be achieved by single conventional controllers.

1.2.3 Neural Network Controller Design Process

The process of designing a neural network-based controller using imitation learning involves the following steps:

1. **Data Collection:** Collect a dataset D of input-output pairs from the expert controller C. This dataset includes system states x, references r, and corresponding control actions u generated by the expert controller:
$$D = \{(x, r, u) \mid u = C(x)\}.$$
2. **Training:** Use the collected dataset to train the neural network. The neural network learns to map the system states x to the control inputs u, effectively replicating the behavior of expert controller C
3. **Evaluation:** Validate the performance of the trained neural network to ensure it performs as well as the expert controller. It is important to formalize how to characterize the imitation quality of the neural network controller. This is done by using the satisfaction degree of the control objectives. The next section describes how we formally describe our control objectives.

1.3 Formal Control Specification

One important aspect of our systems design methodology is to exploit formal methods. The control objectives are rigorously described using formal languages and verified using automated tools. In this section, we recall the formalism that we use for specifying control requirements.

1.3.1 Temporal Logic Background

The literature on temporal logics is vast. The reader is referred to this survey [1] on using temporal logics for validation tasks which are close to our framework. Temporal logics are used to specify patterns that the timed behaviors of systems may or may not satisfy. These logics include basic operators along with temporal operators that allow for the expression of time-dependent behaviors.

Temporal Operators

- **NEXT (X):** Specifies that a condition will hold in the next time step.
- **Always (G):** Specifies that a condition will always hold.
- **Eventually (F):** Specifies that a condition will hold at some point in the future.
- **Until (U):** Specifies that one condition will hold until another condition becomes true.

Types of Temporal Logics

Different types of temporal logics are used based on the nature of the predicates and time:

- **Linear Temporal Logic (LTL) example:**

$$G(r \rightarrow Fg)$$

In LTL the predicates are defined over Boolean signals in discrete/logical time. This above example intuitively reads: always (that is, in every time step), if r is true then eventually g is true

- **Metric Temporal Logic (MTL) example:**

$$G(r \rightarrow F_{[0,0.5s]}g)$$

In MTL the predicates are defined over Boolean signals in real time. This above example intuitively reads: always if r is true then eventually g is true within 0.5 seconds

- **Signal Temporal Logic (STL):**

$$G(x[t] > 0 \rightarrow F_{[0,0.5s]}y[t] > 0)$$

In STL, the predicates are defined over real-valued signals in real time.

Example

Consider the following STL specification:

$$\varphi := G(x[t] > 0.5 \rightarrow F_{[0,1.0]}(G_{[0,1.5]}x[t] < 0.5))$$

This formula specifies that whenever $|x|$ is greater than 0.5, within 0.6 seconds, $|x|$ should settle under 0.5 and remain so for at least 1.5 seconds. Figure 1.2[1] shows an example...

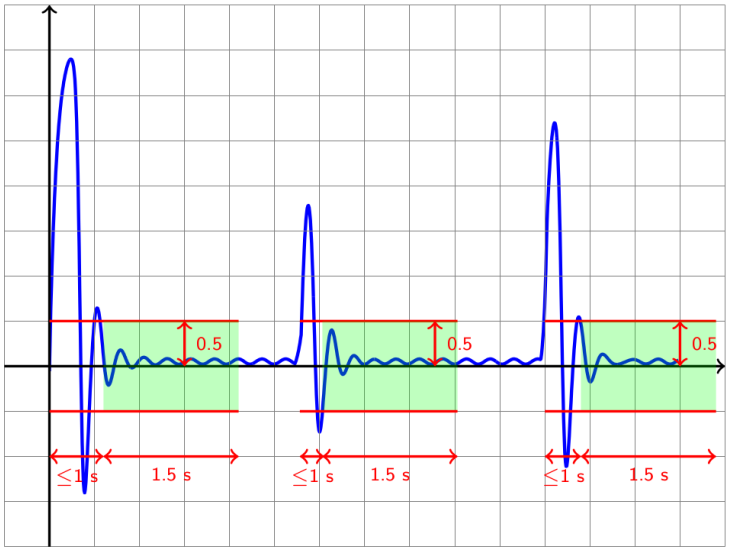

Figure 1.2: STL [4]

1.3.2 Parametric Signal Temporal Logic (PSTL)

PSTL extends STL by allowing the replacement of numeric constants with symbolic variables or parameters.

Parameters: Parameters can be adjusted or learned. For instance, $\varphi = \square_{[0,\tau]}(\|y(t)\| < s)$ uses parameters τ and s.

[1] This figure is taken from https://people.eecs.berkeley.edu/~sseshia/fmee/lectures/EECS294-98_Spring2014_STL_Lecture.pdf

Valuation: A PSTL formula is concretized into an STL formula through a valuation (mapping symbolic parameters to real numbers). For example, $p : \{\tau \to 2, s \to 10\}$ makes $\varphi(p) = \square_{[0,2]}(\|y(t)\| < 10)$.

Monotonicity: Some PSTL formulas can be monotonic, meaning their satisfaction increases with certain parameter values meaning their satisfaction degree increases with respect to parameter values

Use: Suitable for scenarios where the specific numeric thresholds or timing constraints are not known *a priori* and need to be inferred or optimized.

1.4 Formal Validation – Breach Tool

Breach is a MATLAB/C++ toolbox designed for time series analysis and simulation-based analysis of dynamical, Cyber-Physical Systems (CPS), and hybrid systems [4]. This tool is crucial for handling validation tasks that involve Signal Temporal Logic (STL). In our framework, we use this tool to validate neural network controllers and generate data from eventual counter-examples.

Key Functionalities:

- **Time Series Handling**: Breach allows for the import and management of time series data as traces, where each trace is a collection of signals indexed by time. The tool provides capabilities to check these time series against specific properties, making it invaluable for ensuring system reliability.
- **STL Monitoring and Verification**: Users can define STL formulas that describe desired system behaviors or properties. Breach then evaluates these formulas against the time series data to determine whether the properties are satisfied. This process is enhanced by Breach's ability to compute the robustness of a formula, which quantifies how strongly a system's behavior meets or deviates from the specified properties.
- **Simulation and Testing**: Breach integrates with Simulink, enabling extensive testing through multiple simulations (e.g., parameter sweeps) and allowing users to quickly browse through results. This feature is particularly useful for model verification and falsification, where Breach uses optimization algorithms to find scenarios that might violate a system's requirements.
- **Falsification**: The toolbox supports the automatic generation of test cases designed to find violations of STL properties, aiding in the identification of potential weaknesses in system designs. This process leverages the concept of robustness[5], which quantifies how well a system's behavior aligns with specified temporal properties. Robustness provides a numerical measure of satisfaction, indicating not only whether a property is satisfied but also how close a system is to violating it. A positive robustness value suggests compliance, while a negative value indicates a violation, with the magnitude reflecting the severity of the failure. By generating test cases that target scenarios with low robustness, the toolbox effectively identifies weaknesses in the system design, allowing engineers to improve the system by addressing potential violations before deployment.

1.5 Case Study - Water Tank System

Choice of the Water Tank System

The water tank system was chosen as the benchmark for this project due to its simplicity and well-understood dynamics. One primary objective of this project was to establish a connection between the Breach toolbox and ROS2, making it crucial to select a case study that allows us to focus on the integration rather than the complexities of the system itself. The water tank model, with its well-understood behavior and clear physical interpretation, served as an ideal candidate for testing and demonstrating the integration of Breach and ROS2.

The water tank system is a classic example in control theory, often used to teach and test fundamental control concepts. It provides a simple, yet effective platform to test the robustness of control algorithms, making it perfect for the falsification process we wanted to implement.

1.5.1 Dynamics of the System with PID Controller

The dynamics of the water tank system in this project are governed by the following differential equation:

$$\frac{dH(t)}{dt} = \frac{b}{A}u(t) - \frac{a}{A}\sqrt{H(t)} \tag{1.1}$$

where:

- $H(t)$ is the water level in the tank at time t,
- $u(t)$ is the control input, which in this case is the inflow rate controlled by the PID controller,
- a and b are system constants that represent the outflow coefficient and the proportionality constant between the control input and inflow rate, respectively,
- A is the cross-sectional area of the tank.

1.6 Counter-Example Guided Imitation Learning Algorithm

One major element in our algorithm is falsification. Its goal is to identify scenarios where the NN fails to meet the desired specifications, defined by Signal Temporal Logic (STL) formulas. This process involves searching for specific reference inputs or initial conditions under which the NN does not satisfy the STL requirements. The behaviour of the systems that does not satisfy the STL requirements are called *counter-examples*[3].

If counter-examples are found, they highlight the weaknesses or limitations of the current NN model. These counter-examples can then be used to generate additional training data, ensuring that the NN learns from its mistakes and improves its performance in subsequent iterations.

The algorithm has the following major steps.

1. **Initial Setup:**
   - Start with an empty dataset and an untrained NN.
2. **Data Generation:**
   - Use the expert controller to generate initial traces from a grid-based sampling of the state space.
   - Filter these traces to ensure ε-separation, which maintains a diverse and well-distributed dataset.
3. **NN Training:**
   - Train the NN using the collected dataset.
   - The NN is trained to minimize the error (typically Root Mean Square Error, RMSE) between its outputs and the outputs of the nominal controller.
4. **Falsification:**
   - Test the NN by searching for counter-examples where the NN fails to satisfy the STL formula.
   - If counter-examples are found, generate additional training data around these scenarios.
5. **Dataset Augmentation:**
   - Incorporate the new data from counter-examples into the existing dataset.
   - Retrain the NN with the augmented dataset.
6. **Iteration and Termination:**
   - Repeat the data generation, NN training, and falsification steps until no new counter-examples are found or a maximum number of iterations is reached.

- If no counter-examples are found, the NN is considered to have successfully learned the required behavior.

7. **Final Validation:**

   - The final NN is validated to ensure it meets the performance criteria with sufficient coverage and robustness.

Chapter 2

Multiple Controller Imitation Learning

In this chapter, we apply the formal imitation learning framework to combine multiple expert controllers with different characteristics, in order to create a more optimal control strategy. This hybrid control scheme serves as the basis for training a neural network[10] to mimic the combined behavior of multiple controllers, aiming to achieve a balance between speed and precision in the control process. We will consider PID controllers as expert controllers for our case study.

2.1 Context and Motivation

In traditional control systems, designing a controller often involves significant trade-offs between conflicting objectives, such as speed, precision, and stability. Conventional PID controllers, though widely used due to their simplicity and effectiveness, have inherent limitations. A single PID controller may struggle to achieve an optimal balance between these objectives, especially in systems with complex or varying dynamics.

The primary motivation for our approach is to address these limitations by leveraging the strengths of multiple PID controllers with distinct characteristics. In particular, we focus on combining two PID controllers: one optimized for fast response and the other for high precision. Each controller operates effectively within a specific domain but has limitations outside its optimal operating range. For instance, the fast controller may achieve quick responses but risks introducing overshoot and instability, while the slow controller provides stability at the cost of slower reaction times.

The conflict between these two controllers arises because they are optimized for different objectives, and no single controller can excel in all scenarios. By combining these controllers, we aim to create a hybrid control strategy that dynamically switches between them based on the system's current requirements. This approach allows us to capitalize on the fast controller's ability to respond quickly when needed, while also ensuring that the system remains stable and precise through the slow controller.

Moreover, this hybrid control scheme forms the foundation for training a neural network through imitation learning. The neural network is trained to mimic the behavior of the combined controllers, learning when to favor speed and when to prioritize precision. This results in a more adaptive and optimal control strategy that outperforms any single PID controller, addressing the inherent limitations of conventional control methods.

2.2 Control requirements

Normally, the STL is used to define requirements before designing the two PIDs. We rewrite this paragraph as follows: For the water tank case study, we are interested in two requirements, namely overshoot and stabilisation time, expressed using Signal Temporal Logic (STL), as shown in the the following.

For the water tank case study, we are particularly interested in two performance requirements, namely overshoot and stabilization time. These requirements are expressed using Signal Temporal Logic (STL),

which allows us to formally define and monitor the desired behavior of the system. The STL formulas help ensure that the system maintains optimal control performance with respect to these key metrics.

2.2.1 STL requirements

To monitor overshoot and convergence time, the following STL[7] formulas were employed. The overshoot and stabilisation requirements are expressed as follows.

- **Parameter Definitions**:

$$\begin{aligned} &\mathrm{dt} = 0.001, \\ &T_1 = 6, \\ &T_2 = 4, \\ &\mathrm{ov} = 0.02, \\ &d = 0.01 \\ &ts = 4 \end{aligned}$$

- **Stability Formula ($\phi_{\mathbf{stab}}$)**: This formula ensures that if the reference signal changes by more than a certain threshold d, the system's output $H[t]$ will converge to within a small margin of the reference signal $ref[t]$ within a specified time interval, known as the stability time ts. The formula is defined as:

$$\phi_{\text{stab}} := \text{alw}_{[0,10]} \left((|\text{ref}[t + \text{dt}] - \text{ref}[t]| > d) \vee (\text{ref}[t] = 0) \right) \rightarrow \left(\text{ev}_{[0,ts]} \left(\text{alw}_{[0,1]} \left(|\text{H}[t] - \text{ref}[t]| < 0.2 \right) \right) \right)$$

- **Overshoot Formula ($\phi_{\mathbf{ov}}$)**: This formula checks that the system does not exceed a specified overshoot limit (ov) relative to the reference signal $ref[t]$ over time. The formula is written as:

$$\begin{aligned} \phi_{\text{ov}} := &\left(\text{ref}[0] - \text{H}[0] \geq 0\right) \rightarrow \text{alw}_{[T_1,T_2]} \left(\text{H}[t] - \text{ref}[t] < \text{ref}[t] \times \text{ov}\right) \\ &\wedge \left(\text{ref}[0] - \text{H}[0] < 0\right) \rightarrow \text{alw}_{[T_1,T_2]} \left(\text{ref}[t] - \text{H}[t] < \text{ref}[t] \times \text{ov}\right) \\ &\wedge \text{alw}_{[0,10-\text{dt}]} \left(\text{ref}[t + \text{dt}] - \text{ref}[t] > d\right) \rightarrow \text{alw}_{[T_1,T_2]} \left(\text{H}[t] - \text{ref}[t] < \text{ref}[t] \times \text{ov}\right) \\ &\wedge \text{alw}_{[0,10-\text{dt}]} \left(\text{ref}[t] - \text{ref}[t + \text{dt}] > d\right) \rightarrow \text{alw}_{[T_1,T_2]} \left(\text{ref}[t] - \text{H}[t] < \text{ref}[t] \times \text{ov}\right) \end{aligned}$$

- **Combined Formula (ϕ)**: The overall robustness of the system is tested by combining the stability and overshoot formulas:

$$\phi := \phi_{\text{stab}} \wedge \phi_{\text{ov}}$$

2.3 Multiple Controller Imitation Strategy

To implement this approach, the following steps were taken:

1. **Designing Two PID Controllers**:
   - **Fast PID Controller**: This controller is tuned for a quick response, prioritizing speed. However, this comes with the drawback of potential overshoot, which may lead to instability in some situations.
   - **Slow PID Controller**: This controller is designed to avoid overshoot, focusing on stability and precision. The trade-off here is a slower response time.
2. **Data Generation From Multiple Controller Combination**:
   - Our goal is to generate data so that the neural network can imitate the best from all the controller behaviours at every step. To this end, in each step, one needs to decide which controller to imitate, or in other words the control action of which controller to keep in the dataset. In case of the two PID (fast and slow) a mechanism is devised to pick one controller

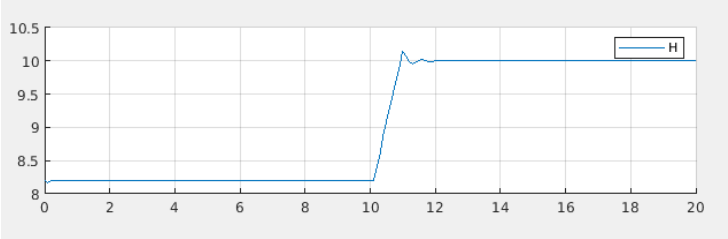

Figure 2.1: Evolution of the water tank level over time using fast PID

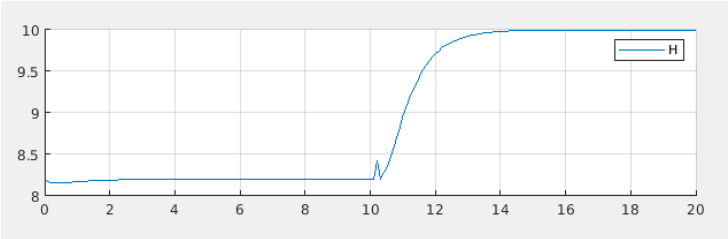

Figure 2.2: Evolution of the water tank level over time using slow PID

based on the current state of the system and the satisfaction degree of the STL requirements. In our case study, the STL specifications, described in the next section, concern stabilisation time and overshoot. A selection strategy could thus be based on criteria such as the magnitude of the error, the rate of change of the error, or specific system conditions that favor one controller over the other.

- The switching logic resulting from the selection strategy is crucial and must be designed to avoid introducing instability or excessive oscillations due to frequent switching.

Algorithm 1 Data Generation from Controller Combination

if $ref_0 > H_0$ **then**
    $threshold_H \leftarrow ref_0 + Oshoot$
    $threshold_L \leftarrow -\infty$
else
    $threshold_L \leftarrow ref_0 - Oshoot$
    $threshold_H \leftarrow \infty$
end if
if Hf exceeds $threshold_H$ **or** Hf is below $threshold_L$ **then**
    $U \leftarrow U_s$
    $HOUT \leftarrow H_s$
else if Close to overshoot boundary then
    if $|Hf - ref| < |Hs - ref|$ **and** Difference is significant **then**
        $U \leftarrow U_f$
        $HOUT \leftarrow H_f$
    else
        $U \leftarrow U_s$
        $HOUT \leftarrow H_s$
    end if
else
    if $|Hf - ref| < |Hs - ref|$ **then**
        $U \leftarrow U_f$
        $HOUT \leftarrow H_f$
    else
        $U \leftarrow U_s$
        $HOUT \leftarrow H_s$
    end if
end if

3. **Training a Neural Network to Imitate the Combined Controller**:
    - With the two PID controllers operating in tandem, the behavior of the combined system is observed and used as the target for training a neural network.
    - The neural network is trained to imitate the combined behavior of the two PID controllers, effectively learning a control strategy that incorporates both speed and precision.
    - The training process involves using data from the combined PID system to adjust the neural network's parameters so that it can replicate the desired control actions.

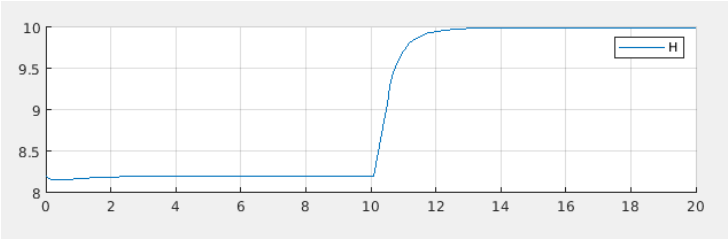

Figure 2.3: Evolution of the water tank level over time using combined PID

2.4 Verification of Signal Temporal Logic (STL) Requirements

Once the neural network has been trained to imitate the combined PID controller, the next step is to verify its robustness.

2.4.1 Verification Process

Using the STL formulas defined above, the robustness of the neural network-based controller was evaluated. The process involved:

1. **Baseline Verification**: Initially, the real system's robustness was verified using the combined formula ϕ, which served as a baseline for comparison.
2. **Neural Network Verification**: The same formula was then applied to the neural network-based controller to assess whether it meets or exceeds the robustness criteria established by the real system.
3. **Iterative Refinement**: If the neural network failed to satisfy the robustness properties, the training process was revisited, and the network was retrained until it met the desired performance standards.

This verification approach allowed us to rigorously evaluate the neural network's performance in imitating the combined PID controller, ensuring that it could effectively balance speed and stability while avoiding excessive overshoot.

- **Performance Gains**: The alternating controller strategy allows the system to respond quickly when needed, while still maintaining precision and stability in steady-state conditions.
- **Neural Network Effectiveness**: The neural network's ability to learn and replicate this strategy suggests that it can generalize well to similar control problems, offering a flexible solution to balancing competing control objectives.

2.5 Results and Discussion

The combined PID imitation approach, when successfully implemented and mimicked by the neural network, has the potential to significantly enhance the performance of control systems. The neural network, trained to learn from both a fast and a slow PID controller, can achieve a balance between rapid response and stable, overshoot-free control.

```
*****************************************
Starting Nelder Mead optimization from x0:
        H_0 = 9.45
        ref_0 = 9.26914
        Vp_0 = 0.448

#calls (max:  300)        time spent (max: Inf)     [current  obj]     (current best)
       250                      157.5               [+1.00000e-02]     (+1.00000e-02)
       260                      163.9               [+1.00000e-02]     (+1.00000e-02)
       270                      170.2               [+1.00000e-02]     (+1.00000e-02)
       280                      176.5               [+1.00000e-02]     (+1.00000e-02)
       290                      182.8               [+1.00000e-02]     (+1.00000e-02)
Best value found during local phase: 0.01 with
        H_0 = 9.42372
        ref_0 = 9.26914
        Vp_0 = 0.448


TEST QUASI-RANDOM SAMPLES


++++++++++++++++++++++++++++++++++++++++++++
Running 10 quasi-random samples with seed 51
#calls (max:  300)        time spent (max: Inf)     [current  obj]     (current best)
       300                      189.1               [+1.00000e-02]     (+1.00000e-02)

END OPTIMIZATION METAHEURISTICS

 Stopped after max_obj_eval was reached (maximum number of objective function evaluations).
No falsifying trace found.
worst_rob:0.01, avg_rob: 0.0122553, #cex: 0
SUCCESS.
>>
```

Figure 2.4: Result of training.

Future work could explore the application of this approach to other types of controllers or more complex systems, as well as further refinement of the multiple controller data selection strategy and STL verification methods.

Chapter 3

Implementation with ROS2

ROS2[8] is a powerful middleware designed for robot interfaces, enabling seamless communication between different components of a robotic system. In this project, ROS2 plays a crucial role in interfacing with the Breach toolbox, which we uses to train and test the robustness of our neural network using Signal Temporal Logic (STL) criteria. ROS2 is in charge of producing data in form of simulation traces to Breach.

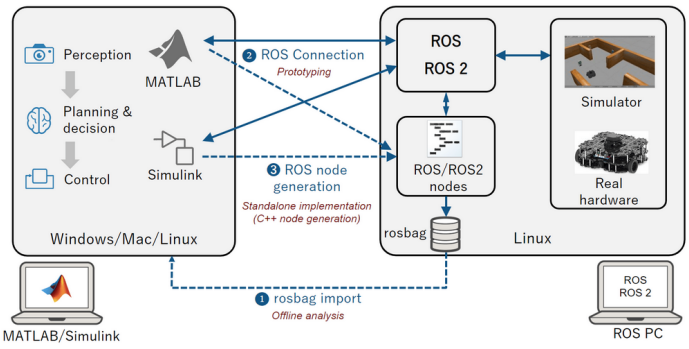

Figure 3.1: connection between ROS2 and MATLAB, sourced from https://fr.mathworks.com/help/ros/index.html?s_tid=CRUX_lftnav.

3.1 Objective

The primary objective was to establish an interface between ROS2 and the Breach toolbox. Given that our algorithm operates offline, real-time data acquisition is not necessary. Instead, we focused on gathering traces of data that could be fed into the neural network training. After training, and if a counterexample is found during the falsification process, we will return to the system with the expert controller to provide data for the neural network to imitate.

3.2 Interest of ROS2

ROS2 offers several advantages that are particularly relevant to this project:

- **Real-Time Uncertainty Handling**: ROS2 is designed with real-time communication in mind, which is essential when dealing with the inherent uncertainties of real-time systems. This capability was crucial when testing the robustness of our neural network.
- **Node Communication**: ROS2 provides a robust framework for inter-node communication, allowing us to efficiently manage the data exchange between different parts of the system. This was particularly important for coordinating the simulation and control nodes.

- **Data Format and Collection Interface**: ROS2 supports various data formats, which facilitated the integration with MATLAB and the Breach toolbox. Additionally, its flexible interface for data collection, including tools like rosbag, made it easier to log and retrieve the data needed for our analysis.

3.3 Simulation and Data Generation Using ROS2

Node Structure

The system is structured around two main ROS2 nodes:

- **Controller Node**: Responsible for sending control commands to the system.
- **Simulation Node**: Responsible for running the simulation of the water tank system.

Synchronization between these nodes was essential, as we need to ensure that each command sent by the controller was received before the next simulation iteration. To achieve this, we configured the system for reliable communication by adjusting the Quality of Service (QoS) settings and ensuring a sufficient buffer size to store messages. This setup helped maintain synchronization without significant delays or message loss, which could otherwise affect system performance.

Simulation and Data Generation for Neural Network Training

To effectively train our neural network, we required a large dataset generated from multiple simulations. Initially, we considered running multiple simulations concurrently by utilizing ROS2's parameterization feature. However, during early trials with two water tanks, it became evident that this approach had limitations. The system exhibited slight variations in response times, which were attributed to the high memory demands of running simultaneous simulations. As a result, this method proved inefficient for gathering large-scale, reliable data. We can see the results of 2 simultaneous water tanks in the following figure 3.2

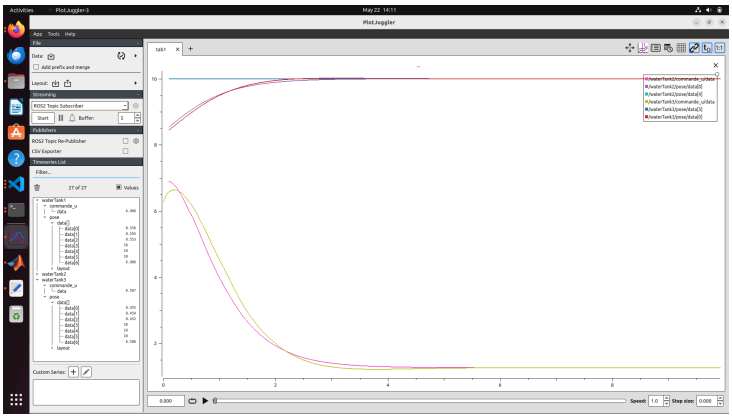

Figure 3.2: simultaneous watertank

Recognizing these constraints, we decided that the simulations would be executed consecutively, rather than concurrently. In MATLAB, we followed a similar strategy, generating quasi-random parameters for each simulation run to ensure a diverse and informative dataset for the neural network. These parameters represented key system variables, such as the initial water tank level and the reference target that the controller aimed to achieve. This approach allowed us to cover a wide range of cases, providing more comprehensive data for training.

The workflow was structured as follows: MATLAB generated the quasi-random parameters, which were then passed into the ROS2 simulation environment. The ROS2 nodes were launched using the function `launch_nodes(H0, ref0)` 3.3, where `H0` represents the initial water tank level and `ref0` represents the desired reference level for the controller. This process ensured that each simulation had unique initial conditions, enriching the diversity of the collected traces.

```
function [t,X, p, status] = sim_breach_watertank(control_fn, t, p)
    %% collect parameters
    DimX = 9;                % last one is cost, which we use with MPC, will see for PID...
    params = p(DimX+1:end); % p(1:DimX) is always 0 for legacy reasons
    Xinit = params(1:7);     % Initial state
    ref1  = params(8);    % Additional parameter(s)

    H0=params(1);
    ref0=params(8);

    %% Init signal output
    X = zeros(DimX, numel(t)); % preparing the signals output array
    X(1:7, 1) = Xinit; % Initial state

    Ts = t(2)-t(1);

    %% simulation

    %Set your desired path for bags
    desiredPath = '/home/sekars/Documents/stage/stl-control-imitation/bags';
    currentDir = pwd;
    cd(desiredPath);

    %launching simulation
    launch_nodes(H0,ref0);

    % return to current path
    cd(currentDir);
    % chemin de fichier .csv
    pose_csv = '/home/sekars/Documents/stage/stl-control-imitation/bags/Info_for_traces/_waterTank1_pose.csv';

    X = downsample_and_format_data(pose_csv);



    status = 0; % required by latest Breach
end
```

Figure 3.3: launch_file

Data Conversion

Data collected via rosbag was initially stored in a .db3 file format. Although we considered using this format directly in MATLAB, it became clear that converting the data to a .csv format was more practical, given that neural network training was performed offline. This conversion facilitated smoother data handling in MATLAB.

The conversion process was automated using the script db3_csv.py 3.4.

```
import pandas as pd
from rosbags.rosbag2 import Reader
from rosbags.typesys import Stores, get_typestore

def bag_to_csv(bag_folder, csv_folder):
    typestore = get_typestore(Stores.ROS2_HUMBLE)  # Utiliser le typestore pour ROS 2 Humble
    with Reader(bag_folder) as reader:
        reader.open()
        for connection in reader.connections:
            topic = connection.topic
            data = []
            for _, timestamp, rawdata in reader.messages(connections=[connection]):
                msg = typestore.deserialize_cdr(rawdata, connection.msgtype)
                row = [timestamp / 1e9]  # Convert timestamp to seconds
                if connection.msgtype == 'std_msgs/msg/Float64MultiArray':
                    row.extend(msg.data)
                elif connection.msgtype == 'std_msgs/msg/Float64':
                    row.append(msg.data)
                data.append(row)

            if connection.msgtype in ['std_msgs/msg/Float64MultiArray', 'std_msgs/msg/Float64']:
                if connection.msgtype == 'std_msgs/msg/Float64MultiArray':
                    columns = ['time'] + [f'data{i}' for i in range(len(msg.data))]
                elif connection.msgtype == 'std_msgs/msg/Float64':
                    columns = ['time', 'data']
                df = pd.DataFrame(data, columns=columns)
                df.to_csv(f"{csv_folder}/{topic.replace('/', '_')}.csv", index=False)

bag_folder = 'rosbag_simu_1'  # Chemin du dossier contenant le fichier .db3 et metadata.yaml
csv_folder = 'Info_for_traces'
bag_to_csv(bag_folder, csv_folder)
```

Figure 3.4: ROSbag conversion

Simulation Parameter Optimization

Throughout the development process, we encountered several challenges that required careful testing and optimization:

Integration Step: One of the recurring issues was determining the best integration step h to balance accuracy and performance. A smaller integration step would generate more accurate data but increase computational load, while a larger step would reduce the data rate but potentially affect simulation fidelity.

Buffer Size: Another issue involved selecting the optimal buffer size for message storage. A larger buffer prevented message loss and ensured reliable communication between nodes, but it also risked consuming too much memory, slowing down the system. After several iterations of testing, we found a balance that ensured reliable communication without overloading the memory.

We optimized the system with the following parameters:

- Final time $T_f = 20$
- Sampling time $T_s = 0.1$

Initially, we used a smaller sampling time of $T_s = 0.001$ to capture more detailed data, but later decided to downsample every 100 steps to achieve a more manageable $T_s = 0.1$. This step was crucial to reduce the volume of data while maintaining the quality needed for training the neural network.

3.3.1 Down-sampling

Down-sampling was introduced as a key solution to handle the vast amount of data generated when using a smaller integration step. While a small time step ($T_s = 0.001$) was essential for capturing the fine details needed for accurate neural network training, it produced an overwhelming volume of data for practical processing. By down-sampling every 100 iterations, we maintained the integrity of the critical data while significantly reducing the total volume of stored information.

One critical aspect of the down-sampling process was to maintain the temporal continuity of the data. For effective neural network training, it was essential to preserve not only the current state but also the previous and previous-previous states for each iteration. This meant that during down-sampling, we had to carefully select which data points to retain, ensuring that the transitions between states were still captured accurately, even when intermediate data was discarded.

This approach allowed us to ensure that the neural network received high-quality inputs, with sufficient temporal context, while avoiding an overload of unnecessary information. As a result, the down-sampled data remained effective for training, providing the necessary precision without excessive computational overhead.

The down-sample process was done using the script down-sample_and_format_data.m 3.5

PIdWatertank_model > C downsample_and_format_data.m

```
function X = downsample_and_format_data(pose_csv)
    % Read the data from the CSV file
    data = readtable(pose_csv);

    % Extract columns assuming the CSV file has columns: 'time' and 'data0' to 'data8'
    time = data.time;
    values0 = data.data0; % H
    values1 = data.data1; % Hp
    values2 = data.data2; % Hpp
    values3 = data.data3; % ref
    values4 = data.data4; % refp
    values5 = data.data5; % refpp
    values6 = data.data6; % V
    values7 = data.data7; % Vp
    values8 = data.data8;

    % Define the downsampling factor
    downsampling_factor = 100;

    % Initialize downsampled arrays
    downsampled_time = [];
    downsampled_values0 = [];
    downsampled_values1 = [];
    downsampled_values2 = [];
    downsampled_values3 = [];
    downsampled_values4 = [];
    downsampled_values5 = [];
    downsampled_values6 = [];
    downsampled_values7 = [];
    downsampled_values8 = [];

    % Loop through the data with an appropriate offset
    for i = 1:downsampling_factor:10100
        downsampled_time = [downsampled_time; time(i)];
        downsampled_values0 = [downsampled_values0; values0(i)];
        downsampled_values3 = [downsampled_values3; values3(i)];
        downsampled_values6 = [downsampled_values6; values6(i)];

        if i > downsampling_factor
            % Ensure that Hp of current downsampled point is H of previous downsampled point
            downsampled_values1 = [downsampled_values1; values0(i - downsampling_factor)];
            % Ensure that refp of current downsampled point is ref of previous downsampled point
            downsampled_values4 = [downsampled_values4; values3(i - downsampling_factor)];
            % Ensure that Vp of current downsampled point is V of previous downsampled point
            downsampled_values7 = [downsampled_values7; values6(i - downsampling_factor)];
```

```
        if i > downsampling_factor
            % Ensure that Hp of current downsampled point is H of previous downsampled point
            downsampled_values1 = [downsampled_values1; values0(i - downsampling_factor)];
            % Ensure that refp of current downsampled point is ref of previous downsampled point
            downsampled_values4 = [downsampled_values4; values3(i - downsampling_factor)];
            % Ensure that Vp of current downsampled point is V of previous downsampled point
            downsampled_values7 = [downsampled_values7; values6(i - downsampling_factor)];

            if i > 2 * downsampling_factor
                % Ensure that Hpp of current downsampled point is Hp of previous downsampled point
                downsampled_values2 = [downsampled_values2; values0(i - 2 * downsampling_factor)];
                % Ensure that refpp of current downsampled point is refp of previous downsampled point
                downsampled_values5 = [downsampled_values5; values3(i - 2 * downsampling_factor)];
            else
                downsampled_values2 = [downsampled_values2; values0(1)]; % Use first value of H
                downsampled_values5 = [downsampled_values5; values3(1)]; % Use first value of ref
            end
        else
            downsampled_values1 = [downsampled_values1; values0(1)]; % Use first value of H
            downsampled_values2 = [downsampled_values2; values0(1)]; % Use first value of H
            downsampled_values4 = [downsampled_values4; values3(1)]; % Use first value of ref
            downsampled_values5 = [downsampled_values5; values3(1)]; % Use first value of ref
            downsampled_values7 = [downsampled_values7; values6(1)]; % Use first value of V
        end

        downsampled_values8 = [downsampled_values8; values8(i)];
    end

    % Combine downsampled data into a table
    downsampled_data = table(downsampled_time, downsampled_values0, downsampled_values1, downsampled_values2, downsampled_values3,...
     downsampled_values4, downsampled_values5, downsampled_values6, downsampled_values7, downsampled_values8, ...
        'VariableNames', {'time', 'Value0', 'Value1', 'Value2', 'Value3', 'Value4', 'Value5', 'Value6', 'Value7', 'Value8'});

    % Convert table to array
    X = table2array(downsampled_data);

    % Transpose the array and remove the first row (time)
    X = X';
    X(1, :) = [];
end
```

Figure 3.5: downsample code

3.3.2 ROSbag Utility and Data Collection

The use of rosbag[6] posed its own set of challenges. Initially, we planned to perform multiple consecutive simulations with rosbag directly integrated into the launch file. However, this approach proved unreliable, as rosbag would often start recording too late, missing valuable data from the early stages of the simulation. we can see this problem in the following Figure 3.6

```
[NodeSim-1] [INFO] [1724675358.554212296] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999939
[NodeSim-1] [INFO] [1724675358.555206556] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999955
[NodeSim-1] [INFO] [1724675358.556212051] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999966
[NodeSim-1] [INFO] [1724675358.557207066] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999974
[NodeSim-1] [INFO] [1724675358.558207063] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999980
[NodeSim-1] [INFO] [1724675358.559208788] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999984
[NodeSim-1] [INFO] [1724675358.560208338] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999988
[NodeSim-1] [INFO] [1724675358.561249108] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999990
[NodeSim-1] [INFO] [1724675358.562248413] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999992
[NodeSim-1] [INFO] [1724675358.563211227] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999994
[NodeSim-1] [INFO] [1724675358.564238741] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999995
[NodeSim-1] [INFO] [1724675358.565221130] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999996
[NodeSim-1] [INFO] [1724675358.566207140] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999996
[NodeSim-1] [INFO] [1724675358.567242540] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999997
[NodeSim-1] [INFO] [1724675358.568213261] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999997
[NodeSim-1] [INFO] [1724675358.569200146] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999998
[NodeSim-1] [INFO] [1724675358.570274233] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999998
[NodeSim-1] [INFO] [1724675358.571255819] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999998
[NodeSim-1] [INFO] [1724675358.572201497] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.573208595] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.574257704] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.575260755] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.576252978] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.577273718] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.578193449] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.579194146] [waterTank1.NodeSim]: Publishing: X_[0] = 9.999999
[NodeSim-1] [INFO] [1724675358.580198384] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.581198889] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.582280686] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.583191103] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.584196108] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.585258954] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.586255009] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.587254419] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.588272409] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.589214011] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.590228314] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.591216394] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.592222328] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[INFO] [1724675358.593282697] [rosbag2_recorder]: Subscribed to topic '/waterTank1/commande_u'
[NodeSim-1] [INFO] [1724675358.593228923] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.594212113] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[INFO] [1724675358.595158418] [rosbag2_recorder]: Subscribed to topic '/waterTank1/pose'
[NodeSim-1] [INFO] [1724675358.595224211] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.596253870] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.597264266] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.598229927] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.599206103] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.600232715] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.601263482] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.602264611] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.603252426] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.604192673] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.605255677] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
[NodeSim-1] [INFO] [1724675358.606199450] [waterTank1.NodeSim]: Publishing: X_[0] = 10.000000
```

Figure 3.6: ROSbag delay

To resolve this, we created a custom script that launched the ROS2 nodes first and then started the

rosbag recording. Despite this adjustment, the issue persisted: the rosbag recording would still miss critical data at the start of the simulation. One potential solution could be printing the data directly into a .txt file, ensuring that no information is lost.

3.3.3 Automatisation

The process of automating multiple simulations in ROS2 involved managing both the simulation launch and the data recording processes. Typically, ROS2 simulations are launched via Python scripts, such as `Nodes_launch.py`, with data recorded using rosbag. However, automating the launch of consecutive simulations, while ensuring that each simulation was properly initialized and that its data was accurately recorded, presented challenges.

To address this, we developed a custom script, `automatisation.py`, which streamlined the entire process. This script ensured that each ROS2 simulation was launched sequentially, rather than simultaneously, to avoid the memory issues encountered with concurrent simulations. The script also managed the timing of data recording, launching the nodes and triggering rosbag in the appropriate sequence to avoid losing critical data at the beginning of each simulation.

Parameter settings for each simulation were either defined in a `.yaml` file or passed directly via command-line arguments. After extensive testing, command-line parameterization was found to be the most reliable method, offering better control and flexibility during the automation process.

Before integrating ROS2, the water tank simulation algorithm generated various initial state parameters. These parameters were essential for producing diverse and informative traces, which were then passed on to the ROS2 environment for further simulation. The automation script ensured these parameters were consistently applied to each simulation, maintaining coherence across the dataset used for neural network training.

3.3.4 Conclusion

In this project, we aimed to leverage ROS2 for the simulation and data generation needed to train a neural network for controlling a water tank system. While the system showed potential, we encountered significant challenges that hindered our ability to find the most effective strategies for data collection and processing.

The integration of multiple nodes required careful consideration of synchronization, communication reliability, and data handling. Despite efforts to optimize the simulation parameters, the constraints of running concurrent simulations and effectively using rosbag for data recording led to delays and data loss.

Ultimately, while we successfully implemented various automation strategies and data processing techniques, we were unable to achieve the desired outcomes for our neural network training due to these limitations. Future work will need to focus on refining these methods, possibly exploring alternative frameworks or tools that could better accommodate the requirements of our project. This experience underscores the complexity of system integration and the importance of iterative testing and optimization in achieving successful outcomes in robotic simulations.

Conclusion

In conclusion, this internship has been a deeply enriching experience, both technically and personally. On the technical side, I gained hands-on expertise in advanced tools and methodologies, such as formal verification and falsification of system properties using temporal logic. I became proficient in using toolboxes designed to automatically generate test cases, applying robustness metrics to identify potential system design flaws. This significantly enhanced my understanding of system dynamics, signal processing, and optimization techniques.

Personally, the internship helped me sharpen my problem-solving skills and adapt to a dynamic work environment. I developed better time management, learned how to navigate complex projects, and improved my collaboration and communication skills while working with a team. These experiences have equipped me with a broader perspective on real-world challenges and a stronger foundation to tackle future projects.

Bibliography

[1] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. 2018.

[2] S. Chen, K. Saulnier, N. Atanasov, D. D. Lee, V. Kumar, G. J. Pappas, and M. Morari. Approximating explicit model predictive control using constrained neural networks. In *Annual American Control Conference (ACC)*, pages 1520–1527, 2018.

[3] Thao Dang, Alexandre Donzé, Inzemamul Haque, Nikolaos Kekatos, and Indranil Saha. Counter-example guided imitation learning of feedback controllers from temporal logic specifications. In *62nd IEEE Conference on Decision and Control (CDC)*, Singapore, Singapore, Dec 2023. IEEE. hal-04295795.

[4] Alexandre Donzé. Breach, A toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification, 22nd International Conference*, pages 167–170. Springer, 2010.

[5] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings*, volume 6246 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2010.

[6] Julius Köhler, Karsten Knese, and Dirk Thomas. rosbag2: A file format and framework for robotics data. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7902–7908. IEEE, 2021.

[7] Akshay Mambakam, José Ignacio Requeno Jarabo, Alexey Bakhirkin, Nicolas Basset, and Thao Dang. Mining of extended signal temporal logic specifications with paretolib 2.0. *Formal Methods Syst. Des.*, 62(1):260–284, 2024.

[8] Yutaka Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.

[9] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J Andrew Bagnell, Pieter Abbeel, Jan Peters, et al. An algorithmic perspective on imitation learning. *Foundations and Trends in Robotics*, 7(1-2):1–179, 2018.

[10] Pratyush Varshney, Gajendra Nagar, and Indranil Saha. Deepcontrol: Energy-efficient control of a quadrotor using a deep neural network. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 43–50, 2019.

[11] Yang Wang and Stephen Boyd. Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, 18(2):267–278, 2010.