Online-adaptive PID control using Reinforcement Learning

1st Detlef Arend, 2nd Amerik Toni Singh Padda, 4th Andreas Schwung
Automation Technology and Learning Systems
South Westphalia University of Applied Sciences
Soest, Germany
{arend.detlef, schwung.andreas}@fh-swf.de,
amerik.singh13@gmail.com

3rd Dorothea Schwung
Artificial Intelligence and Data Science
in Automation Technology
Hochschule Düsseldorf University of Applied Sciences
Düsseldorf, Germany
dorothea.schwung@hs-duesseldorf.de

***Abstract*—This paper presents the novel RLPID architecture for online-adaptive control, which combines classical proportional-integral-derivative (PID) control with reinforcement learning (RL). This hybrid approach enables dynamic online adjustment of PID parameters during control operation. Specifically, we propose a multi-objective reward structure that integrates established control criteria and analyze suitable configurations for different system dynamics. The RLPID controller has been implemented within the open-source middleware MLPro, where it is embedded in newly developed sub-frameworks for classical and online-adaptive control. Owing to its hybrid nature, the architecture can be used both in traditional control loops and within the Markov decision process of RL. Its effectiveness and practical applicability are demonstrated in a cascade control scenario.**

***Index Terms*—PID control, online machine learning, online-adaptive control, reinforcement learning, MLPro**

I. Introduction

A. Background and problem statement

Modern industries rely on complex processes that are efficiently and environmentally friendly operated. Advanced control strategies such as model predictive controllers offer effective solutions to achieve these goals [5]. However, the model identification represents a significant challenge for such solutions, in particular, due to the dynamic nature of processes and their inherent complexity. Although these control concepts initially perform satisfactorily, their long-term effectiveness depends heavily on the performance of the PID controllers in the lower control level, whose behavior may change over time.

PID controllers are characterized by their simple structure, stability, and robustness and hence offer a practical solution for many control problems [19]. Their simplicity of only three tunable parameters (K_p, T_n, and T_v) reduces the development effort and often makes them preferred over more complex controllers. These parameters, however, are process-specific and require careful and continuous adjustment to achieve optimum results.

The most basic method for tuning PID controllers is experimentally adjustment based on the system response. Rule-based methods, such as the Ziegler-Nichols [29] method and subsequent developments [9], are based on certain signal characteristics (e.g., decay ratio, settling time) and offer practical approaches for setting parameters. However, such methods push the system into an oscillatory state, which can be problematic in sensitive environments.

Various approaches have been developed to cope with these challenges. Specifically, [20] propose PID auto-tuning based on ON/OFF control, [25] use the Levenberg-Marquart algorithm while [17] employ neuro-fuzzy models. All of these algorithms are designed for specific system structures.

Reinforcement learning (RL) [26] has recently proven itself in numerous applications, including control and supervision [22, 23]. In contrast to other ML approaches based on static data sets, RL considers temporal dependencies and uses delayed rewards to optimize decisions. A particular advantage of RL is its ability to continuously learn and adapt to changing processes, such as fluctuations in input variables or operating conditions. These properties have made RL increasingly valuable in process control and optimization [18].

Recently, few works have employed RL for PID parameter tuning in simulated environments. Specifically, [7] proposes an RL-based tuning with constraints consisting of a system identification step with subsequent fine-tuning of the parameters. RL-based adaptive PID-control is presented in [24] for the control of nonlinear unstable processes. Also, the work [11] specifically concentrates on stability preservation when using PID parameter tuning. A novel RL algorithm based on entropy maximization for automatic tuning of PID parameters has been proposed in [6]. A real-world application of RL-based PID tuning is presented in [13], discussing various implementation aspects on standard hardware.

However, these approaches have considerable limitations: they are often tailored specifically to the systems examined in the respective scenario and are, therefore, difficult to transfer to other use cases. The lack of reusability, modularity, and standardization of the existing approaches not only makes it difficult to apply them in new scenarios but also impacts the comparability and further development of these methods. This represents a fundamental challenge, particularly in an interdisciplinary field such as adaptive control, where

flexibility and scalability are crucial.

B. Proposed solution and contributions

To address the challenges outlined above, we extended our open-source middleware MLPro [2, 3] - a framework for standardized machine learning in Python - by adding dedicated sub-frameworks for both classical and online-adaptive RL-based control. A key enhancement is the integration of a reusable design pattern for an RL-based PID controller, which forms the core subject of this study. In summary, our contribution to the field of online-adaptive control consists of the following components:

- **RLPID controller:** A modular, online-adaptive RL-based PID controller that dynamically tunes K_p, T_n, and T_v based on a novel multi-objective reward structure.
- **MLPro framework integration:** Provision of dedicated sub-frameworks for classical and online-adaptive control, delivering the RLPID architecture as a reusable control component.
- **Experimental validation:** Demonstration of adaptability and stability in a cascade control scenario, showcasing real-world applicability.

C. Outline

The remainder of this paper is organized as follows: Section II provides the theoretical background. Section III introduces the proposed online-adaptive PID controller. Section IV describes its integration into the MLPro framework. Experimental results are presented and discussed in Section V. Finally, Section VI concludes the paper and outlines directions for future work.

II. Preliminaries

In this section, we present the preliminaries required for the proposed approach.

A. PID control

The PID controller can be described by the following equation

$$u(t) = K_p \left(e(t) + \frac{1}{T_n} \int_0^t e(\tau)\, d\tau + T_v \frac{de(t)}{dt} \right) \tag{1}$$

where $u(t)$ denotes the control signal, $e(t)$ is the control error defined as the difference between reference and process output, K_p is the proportional gain, T_n the integral time constant, T_v the derivative time constant, and τ is the dummy variable of integration.

B. Reinforcement Learning

RL is formulated as a Markov decision process (MDP) where an agent takes actions in its environment and learns from the observed rewards. The agent's objective is to maximize the expected cumulative reward over a given time horizon. Formally, the MDP is described by the tuple $(\mathcal{S}, \mathcal{A}, P, R, p_0)$ with the set of states $\mathcal{S}$, the set of actions $\mathcal{A}$, the transition model $P : S \times A \times S \rightarrow [0\ 1]$, such that $P(s_t, a_t, s_{t+1}) = p(s_{t+1}|s_t, a_t)$ is the transition probability from state s_t to the following state s_{t+1} by applying action a_t, the reward function $R : S \times A \times S \rightarrow \mathbb{R}$, which assigns a reward $r(s_t, a_t, s_{t+1})$ to each state transition $s_t \leftarrow s_{t+1}$, and an initialization probability p_0 of the states.

The agent's action a_t is taken by the agent's policy $\pi : S \rightarrow A$, resulting in the system changing state to s_{t+1} while observing a reward r_t. Hence, an optimal policy that results in the highest possible cumulated reward is sought by interacting with the environment. Correspondingly, we want to maximize the return $\rho^\pi = E[R_t|\pi]$ with the discounted reward $R = \sum_t \gamma^t r_t$, where $0 \leq \gamma \leq 1$ is the discount factor.

For the above problem, various algorithms have been developed, ranging from simple Q-learning [26] to policy gradient [27] or actor-critic approaches like A2C [16], SAC [8], DDPG [14], and PPO [21]. Policy gradient methods are the basis of actor-critic algorithms where a parameterized randomized policies $\{\pi(s, .|\theta), s \in \mathcal{S}, \theta \in \mathrm{R}^{d_1}\}$ are defined. The gradient of the average reward with respect to the policy parameters θ

$$J(\theta) = \lim_{T\rightarrow\infty} \frac{1}{T} \sum_{t=1}^{T} \rho^\pi \tag{2}$$

is estimated from the observed states, actions, and rewards. Here, ρ^π represents the expected reward obtained when following the policy π, averaged over many possible control situations. It serves as a measure of how well the policy performs in the long run.

Hence, optimal parameters are obtained by

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta J(\theta) \tag{3}$$

with learning rate α_θ. Here, $\nabla_\theta J(\theta)$ represents the gradient of the objective with respect to the policy parameters, indicating the direction of improvement.

By means of the policy gradient theorem [4], this gradient yields

$$\nabla_\theta J(\theta) = \lim_{T\rightarrow\infty} \frac{1}{T} \sum_{t=1}^{T} \sum_{a\in\mathcal{A}} \nabla_\theta \pi(s, a|\theta) Q^\pi(s, a) \tag{4}$$

where

$$Q^\pi(s, a) = \mathbb{E}[\sum_t \gamma^t r_t | s_0, a_0, \pi_\theta] \tag{5}$$

is the state-action value function. Here, $\mathbb{E}[\cdot]$ is the expected value over trajectories under policy π, and $\gamma \in [0, 1]$ is the discount factor weighting future rewards.

Also, the Q-function is parameterized by a neural network, i.e., $Q(s, a|\theta^Q)$. The update of the parameters θ^Q is done based on the standard quadratic temporal difference loss function

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{6}$$

with

$$y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}, a_{i+1}|\theta')|\theta^{Q'}). \tag{7}$$

Here, θ_Q and θ'_Q denote the parameters of the current and target Q-function networks, respectively. The symbols Q' and π' refer to target networks used to stabilize training, whose parameters are updated less frequently than those of the main networks. The variable N in the loss function denotes the mini-batch size, i.e., the number of state-action pairs sampled from the replay buffer to estimate the Q-function loss.

The above-mentioned actor-critic algorithms differ in how the update is designed, the regularization scheme, and whether they are applied online or offline.

III. Online-adaptive RL-based PID control

In this section, we discuss the basic architecture of our novel OA PID controller based on RL, which we will refer to as the RLPID controller.

A. General architecture

The architecture of the RLPID controller shown in Figure 1 essentially consists of two nested main components:

RL controller: The outer component represents a controller that can be classically used within a control loop. The inner structure corresponds to a typical RL approach, which integrates two central elements of a classic RL system: a policy and a reward function. The policy calculates manipulated variables u_t based on control errors e_t. It is trained during operation with the help of the reward function.

RLPID policy: The inner component is a specialized RL policy, which, as such, can be used in both an RL controller and an RL agent. It contains a classic PID controller that calculates the manipulated variable. Its parameters are adjusted during operation by an embedded second RL policy. This, in turn, optimizes its behavior based on the reward provided by the surrounding RL controller. Overall, a policy-in-policy strategy is pursued here.

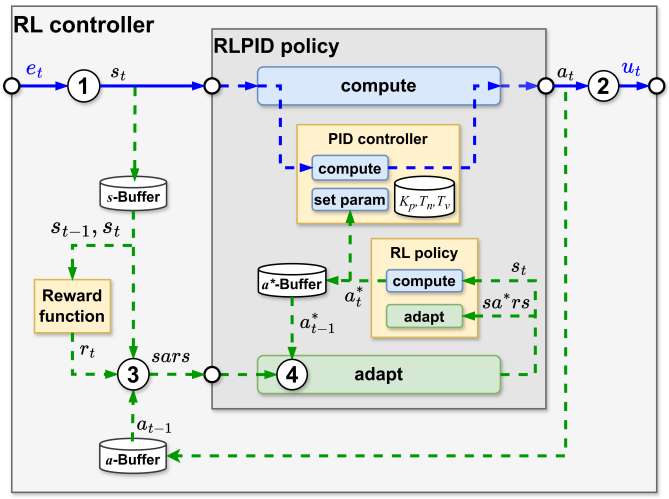

Fig. 1: Architecture of the RLPID controller

B. Internal processes in detail

An incoming control deviation e_t is converted by operator 1 into a status s_t and passed on to the embedded RLPID policy (blue arrows in Figure 1). This uses the integrated PID controller, which carries out the actual calculation of an action a_t based on its current PID parameters $K_{p,t}, T_{n,t}, T_{v,t}$. This action is converted by operator 2 into the manipulated variable u_t and forwarded to a subsequent controlled system.

When adjusting the PID parameters, the reward r_t is first determined by the integrated reward function based on two consecutive statuses s_{t-1}, s_t. Taking into account the action a_{t-1} from the previous run, operator 3 now creates the tuple $sars = (s_{t-1}, a_{t-1}, r_t, s_t)$ required for the RL training. This is passed to the embedded policy for adaptation. A special feature of the RLPID policy presented here is that it internally discards the action a_{t-1} and replaces it with the last action a^*_{t-1} of the inner RL policy. In consequence, a modified tuple $sa^*rs = (s_{t-1}, a^*_{t-1}, r_t, s_t)$ is forwarded to this for adaptation. The updated RL policy is then used to calculate new parameters for the PID controller. The s-/a-/a^*-buffers involved in the adaptation process each store a historical value. The entire adaptation mechanism is marked with green arrows in Figure 1. It should be emphasized that any algorithm, such as Q-Learning, DDPG, or PPO, can be used for the inner RL policy.

In the following, we define the specific requirements for the agent and policy to ensure that the control is both stable and efficient.

State-Action-Space: To use the policy effectively, the action and the state space have to be defined. For the scenario of OA control, the PID parameters K_p, T_n and T_v are therefore the action variables, i.e. $a_t = [K_{p,t}, T_{n,t}, T_{v,t}]$.

To ensure that the actions generated by the policy remain within the physical limits of the system, corresponding value ranges must be defined for each parameter. In general, the limit values depend on the specific application and the controlled system and should, therefore, be flexible and configurable. A typical example is the parameter limits of an industrial PID temperature controller (e.g., JUMO PID controller), which are defined as $K_p \in [0.999]$, $T_n \in [0.999]s$, $T_v \in [0.999]s$.

In addition to the action space, the state space of the policy has to be defined. In RL, the state space defines all occurring states of the environment as well as their physical limits. In the context of OA control, minimizing the control error is the primary goal of the PID controller, so we use $e(t)$ as a single state. Theoretically, $e(t)$ can take values in the range $[-\infty, \infty]$. However, as large values not only pose numerical challenges but can also impair the efficiency of the policy, we normalize as follows

$$e_t = \frac{w_t - y_t}{y_{\max}} \tag{8}$$

where $y_{\max}$ is the maximal expected control value. This normalization maps the control error to the range $e_t \in [-1, 1]$, which reduces the numerical effort and facilitates processing within the policy while the physical meaning of the error is

retained. Consequently, the control values (e_t, e_{t-1}, u_t) are transformed to the RL-values (s_t, s_{t-1}, a_t).

Reward-Function: The reward function is an essential component of any RL architecture and serves as an evaluation benchmark for the agent's performance. In each cycle, it receives the last state s_{t-1} and the current state s_t from the environment. Based on these states s_t and s_{t-1}, a user-specific reward is calculated that either rewards or punishes the agent's actions.

The primary goal is to minimize the current control error e_t as quickly as possible and ideally reduce it to zero. An error-based reward function based on e_t can be formulated as follows:

$$R_0 = -|e_t|. \tag{9}$$

Equation 9 defines a reward based on the current control error e_t. The expression $-|e_t|$ ensures that a negative reward is always generated for $e_t \neq 0$ and that the reward is proportional to the deviation e. However, a major disadvantage is that large errors are not penalized more than small ones, which can lead to a slow correction of larger deviations. To penalize large errors more strongly, the equation 9 can be modified:

$$R_{0*} = -e_t^2. \tag{10}$$

which penalizes larger deviations disproportionately.

The combination of both results in a refined reward function

$$R_1 = R_0 + R_{0*}. \tag{11}$$

In addition to the primary objective of minimizing the control deviation, another important requirement for the reward function is to penalize strong overshoot and persistent oscillations in the control deviation, as is typical in higher-order $PT-n$ systems. To specifically counteract overshoots, the reward function can be extended:

$$R_2 = R_1 - w_3 * \ \max(\ |e_t| - e_{\text{th}}, 0)^2. \tag{12}$$

The additional term $-w_3 * \ \max(\ |e_t| - e_{\text{th}}, 0)^2$ ensures that a penalty is triggered as soon as the control deviation exceeds a defined threshold value e_{th}. Overshooters are penalized disproportionately by the squared difference $(|e_t| - e_{\text{th}})^2$, which motivates the system to minimize such deviations consistently. As long as the control error remains within the defined threshold band, there is no additional penalty.

While the above terms penalize deviation, it might be beneficial to specifically reward the controller for keeping the control error within small limits. This can be encouraged by adding a further term:

$$R_3 = -|e_t| - e_t^2 - \max(|e_t| - e_{\text{th}}, 0)^2 + 30 \cdot \min(|e_t|, 0.5)^2.$$

Stability and Robustness: Maintaining the stability of the control loop is another key requirement for the inner RL policy. When new actions are generated, it must be ensured that they do not lead to unstable system behavior or oscillations. In particular, abrupt and sudden changes to the PID parameters can have a negative impact on system stability. To prevent this, we define the policy so that the changes to the actions take place relative to the previous action:

$$a_t^* = a_{t-1}^* + \Delta a, \text{ for } \Delta a = (a_{t-1}^* - \pi(a_{t-1}^*|s_{t-1})). \tag{13}$$

To prevent abrupt changes to the action, a maximum rate of change Δa_{max} is defined for Δa:

$$|\Delta a| \leq \Delta a_{\text{max}}. \tag{14}$$

However, the introduction of Δa_{max} creates the following dilemma: If the policy generates an action a_t^*, it receives a reward r_t in the next cycle as feedback for the quality of the generated action. Hence, if the PID controller only changes the PID parameters by a defined increment Δa_{max}, the resulting reward r_t from the action a_{t-1}^* does not match. Hence, a solution must be found as to how the policy must behave until the PID controller has gradually brought the new parameters to the target value. One possible solution could be to switch off the adaptation mechanism for the adaptation period and switch it on again via an event. However, it must be decided which reward the policy receives for the action a_t^*. One possible variant is to collect the rewards calculated over the period in a buffer that arise during the adaptation phase. When the adaptation is switched back on, a weighted value of all collected rewards is made available to the policy as $r_{t,mean}$.

Another important aspect is the correct selection of Δa_{max}. If the rate of change is chosen too low, the whole adaptation mechanism becomes too slow. If the rate is too high, there is a risk that a strong change in the PID parameters by the policy cannot absorb the rate of change, and thus the entire control loop may become unstable. Because of the challenges, the introduction should be tested based on experiments.

Adaptation mechanism: The implementation of an OA PID controller requires the consideration of two key aspects namely switchable adaptation and the frequency of adaptation. An OA PID controller operates in the control loop like a classic or an adaptive controller, as there are applications in which continuous adaptation of the controller parameters is not required.

To fulfill this requirement, a trigger logic can be implemented that only activates the adaptation under certain conditions, such as exceeding a defined error limit value or in the event of system changes.

Another crucial aspect is the frequency with which the PID parameters are adapted. Too frequent adaptation can lead to instability, especially in systems where rapid parameter changes could destabilize the system behavior. It is therefore necessary to adapt the adaptation frequency to the specific dynamics of the system. For this purpose, the latency, i.e., dominant time constant t_l, of the controlled system must either be determined or at least known in advance. The general requirement for the adaptation interval Δt_a can be defined as follows:

$$\Delta t_a \geq t_l. \tag{15}$$

This allows the system to react to changes in the policy and continuously optimize the policy based on the system's

response. However, this approach only works if the dynamics of the system are known. Note, that Stable Baseline 3 provides RL algorithms, such as PPO, where the adaptation frequency can be controlled via the hyperparameter n_{steps} where, for $n_{step} > 1$, the data is first collected in an internal buffer over n_{steps} iterations before the adaptation takes place. Alternatively, we introduce a dynamic update, which defines the adaptation steps depending on the current control error:

$$n_{\text{steps}}(e_t, t) = \begin{cases} n_1 & \text{if } e_t \leq \frac{1}{3} e_{\text{norm,max}} \\ n_2 & \text{if } \frac{1}{3} e_{\text{norm,max}} < e_t \leq \frac{2}{3} e_{\text{norm,max}} \\ n_3 & \text{if } e_t > \frac{2}{3} e_{\text{norm,max}} \end{cases} \quad (16)$$

with $t - t_{\text{a-1}} \geq t_{\text{a}}$ and the normalized maximum control error $e_{norm,max}$ serving as a reference measure for controlling the adaptation rate. The condition $t - t_{\text{a-1}} \geq t_{\text{a}}$ ensures that changes to the adaptation frequency are only made after specified time intervals t_a. In the event of a setpoint change, the absolute error $|e_t|$ could increase significantly, which could lead to an unnecessary adjustment of the adaptation rate and thus to the rejection of an optimal parameter set. To avoid this, the equation 16 can be made robust by adding a condition:

$$w(t_{\text{a-1}}) = w(t_a), \text{ then } n_{\text{steps}}(e_t, t). \quad (17)$$

Equation 17 requires that the adaptation frequency is only adjusted if the setpoint has not changed in the period Δt_a. This ensures that stable PID parameters are not unnecessarily modified by the policy in the event of a setpoint change, which could otherwise lead to undesirable system reactions due to the adjusted adaptation rate.

IV. MLPro-Integration of OA control

The approaches considered in the literature for developing online-adaptive PID controllers are often specifically tailored to the systems investigated in the scenario and are, therefore, difficult to transfer to other applications. The lack of reusability, modularity, and standardization of the existing approaches not only makes it challenging to apply them in new scenarios but also to compare and further develop these methods.

This prompted us to add corresponding functionalities to our open-source middleware MLPro [2, 3]. The project was launched in 2021 and was initially designed for hybrid ML applications focusing on RL and Game Theory (GT). Since then, it has been continuously expanded and has included a further sub-framework called MLPro-OA for online machine learning applications since version 2.0. The latest extensions in classical and online adaptive control, particularly the system for RLPID controllers presented here, were provided as part of version 2.1 of MLPro.

As shown in Figure 3, MLPro is organized in different layers. At the lowest layer is the sub-framework MLPro-BF. It provides numerous basic functions, from simple logging and plotting to mathematical functions and the basics of machine learning. It, thus, forms the foundation for all higher ML functionalities in MLPro. As part of our work on version 2.1, another sub-framework, MLPro-BF-Control, was added

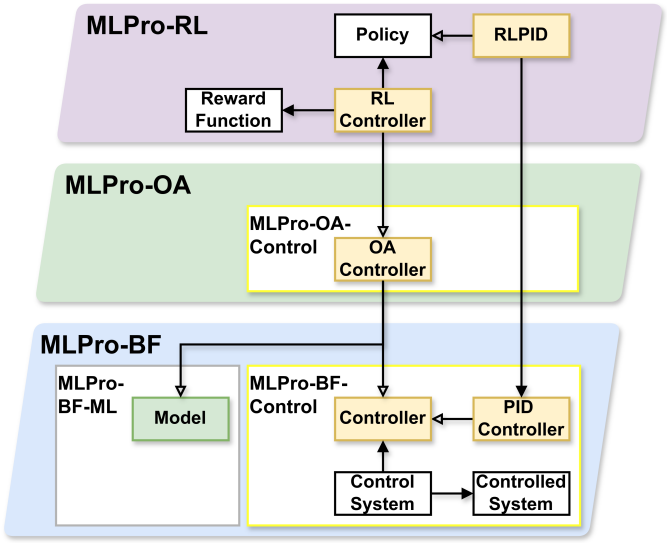

Fig. 2: Integration of OA/RL control in MLPro

for classic control tasks, which we will discuss in more detail in Subsection IV-A. The previously mentioned sub-framework MLPro-OA forms the next higher layer within MLPro. The sub-framework MLPro-OA-Control for online-adaptive control, described in more detail in Subsection IV-B, was added there. The sub-framework MLPro-RL for reinforcement learning, available since version 1.0 of MLPro, was expanded to include hybrid functionalities such as the RL controller and the RLPID policy. We will discuss this in more detail in Subsection IV-C.

We want to point out that this work can only give a rough overview of the developments carried out. For more technical details, program examples, and descriptions, and a complete description of all programming interfaces (API), we refer to the online documentation for MLPro [15].

A. The new sub-framework MLPro-BF-Control

This new sub-framework anchors the classical, non-adaptive closed-loop control in the basic functions of MLPro. The essential classes are

- **Controller:** A template class for custom controller implementations.
- **ControlledSystem:** A template class for custom implementations of controlled systems.
- **ControlSystem:** A container class that combines controllers, controlled systems, and further operators into a control system.
- **PIDController:** The implementation of a classic PID controller as a child class of Controller.

The inheritance relationships between classes are shown as hollow arrows in Figure 3. Filled arrows show usage relationships between classes.

Classes for dynamic systems such as PT1 and PT2 are also included, but not shown.

B. The new sub-framework MLPro-OA-Control

Building on the infrastructure of MLPro-BF, MLPro-OA-Control provides enhanced mechanisms for online-adaptive controllers. At the heart of this is the template class OAController, which combines and integrates the properties of a classic controller (Controller class from the MLPro-BF-Control sub-framework) with those of an ML model (Model class from the MLPro-BF-ML sub-framework) through inheritance. It thus allows the standardized implementation of custom online-adaptive controllers, which can then be integrated into control loops of the underlying MLPro-BF-Control sub-framework.

C. Integration of the RLPID control into the MLPro-RL sub-framework

The functions of the RL controller and the RLPID policy presented in this paper were finally implemented as extensions of the existing sub-framework MLPro-RL. The corresponding classes are

- **RLController:** This wrapper class inherits all properties of the OAController class from the underlying sub-framework MLPro-OA-Control and implements the mechanisms described in Section III. It includes both an RL policy (Policy class) and a reward function (RewardFunction class) in the manner described, whereby both are kept interchangeable.
- **RLPID:** This class implements the RLPID policy introduced in Section III as a child class of the existing Policy class. In particular, it integrates the PID controller implemented in MLPro-BF-Control.

Both of the classes mentioned can be used directly in custom applications. The RLPID class can be used in real RL scenarios as a policy for an RL agent and in classic control applications - embedded in the RLController class - as an online-adaptive controller. Only the reward functions differ in one essential detail: in the RL context, successive system states of an environment are used to calculate a reward, while in control applications, two successive control errors are used for this purpose.

V. Experiments and results

In the following, we demonstrate the capabilities of the RLPID controller in the practical application of cascade control. In Subsection V-A, we describe the experimental setup and the variants our test program went through. We discuss the results in Subsection V-B.

The underlying test program has been made available in a publicly accessible GitHub repository [1]. It illustrates how to apply the MLPro functionalities described in this paper in custom Python programs and enables the reproduction of all experiments.

A. Experimental setup

Each experiment is based on a fixed number of simulation steps, whereby the number of steps and the sampling rate of a control cycle depend on the dominant time constant of the respective controlled system. We particularly focus on the effects of various reward functions and the choice of RL policies. The central criteria include the control performance in terms of stability, overshoots, oscillations, and the efficiency of adaptation, where we consider the speed and stability of the adaptation of PID parameters.

Reward functions: We evaluate the four reward functions R_0, R_1, R_2 and R_3 with $e_{\text{th}} = 3$ as described in Subsection III-B.

RL algorithms: We test various RL algorithms, namely A2C, SAC, DDPG, and PPO, to adapt the PID parameters. We note that we do not focus on optimizing the hyperparameters for each individual policy but rather on illustrating the applicability of RL methods for dynamically adapting the PID parameters. Therefore, the default values from SB3 are used for all policies. Only the learning rate is varied to investigate the influence of this hyperparameter.

PID parameters: The PID controller is initialized with $K_p = 1$, $T_n = 0$ and $T_v = 0$ with value ranges $K_p, T_n, T_v \in [0, 999]$.

Cascade control system: We consider a cascade control problem taken from [10] and shown in Figure 3 where the inner control cascade consists of a P controller with static gain $K_p = 0.38$ combined with a PT1 controlled system characterized by the parameters $K_i = 25$ and $T = 1200s$.

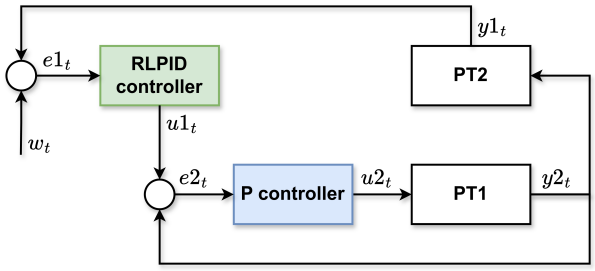

Fig. 3: Experimental setup of a cascade control with RLPID controller

For the outer cascade, the RLPID controller is combined with a PT2 system parameterized as $K_p = 25$ and $T = 1200s$. The setpoint of the system is set to w_t = 40 for the entire duration. A cycle time interval of $t_{cycle,i} = 2s$ is used for the inner control loop to give the system sufficient time to react to changes in the manipulated variable of the PID controller. The outer control loop is updated with a cycle time of $t_{cycle,o} = 4s$.

B. Results

This section presents the results of the RLPID controller embedded in the aforementioned cascade control. We discuss the effectiveness of various reward functions in the control of a PT1-PT2 cascade using a DDPG policy. Then, the performance of the best reward function will be considered in comparison to the other RL algorithms. Finally, we examine which learning rate best suits the system.

Experiment 1 - Reward function for cascaded control: Figure 4 illustrates the control error over the number of control loop cycles for the different reward functions. It can be seen

that a settling time of around 2000 cycles is required before the individual control deviations converge.

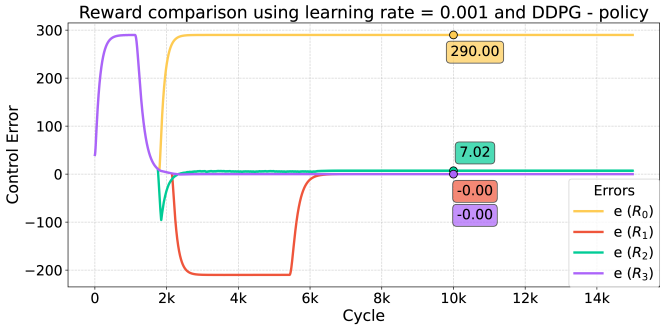

Fig. 4: Experiment 1: Reward comparison using DDPG for cascaded control

As can be seen, reward R_3 delivers the best results, achieving the fastest convergence as well as the lowest control error, and is characterized by its stability and precision. In contrast, R_0 shows no improvement in the control loop and remains constant at a high error level, which makes it unsuitable for use. R_2 is a viable alternative, but exhibits initial instability, which renders the approach unsuitable.

Experiment 2 - RL algorithm for cascaded control: We use R_3 to analyze the performance of different RL algorithms. The results are shown in Fig. 5, showing the control error $e(t)$ over the number of control loop cycles.

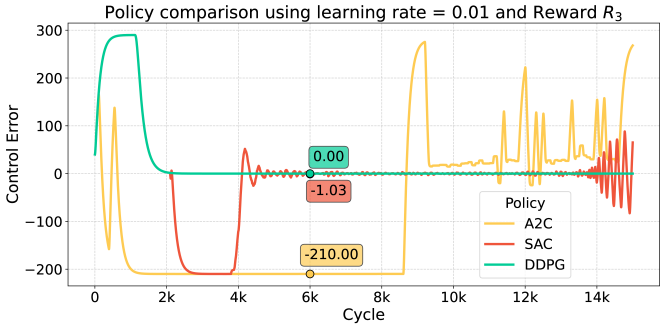

Fig. 5: Experiment 2: Comparison of different RL-algorithms for cascaded control

For clarity, the PPO policy is not shown as the control error oscillated intensely in the range $-300 \leq e(t) \leq 200$, leading to consistently unstable control of the cascade. Also, the A2C policy did not succeed in generating a PID parameter set that allows the control loop to converge to a control error close to 0 or at least ensures a stable fixed control error. In contrast, the SAC policy showed a more stable control behavior during the time interval $4300 \leq e(t) \leq 14000$, with the control error $e(t)$ remaining constantly below 1. However, the SAC policy required a comparatively long settling time of around 400 control loop cycles to find a suitable PID parameter set. From cycle 14,000 onwards, however, the PID parameters were adjusted again, leading to the system oscillating. The DDPG policy, on the other hand, achieved stable control, with a settling time of around 2000 cycles. This settling time could be further reduced through targeted optimization, for example, by fine-tuning the hyperparameters.

Experiment 3 - Learning rate for cascaded control: We further investigate how the learning rate affects the DDPG with R_3. The results are shown in Fig. 6, representing the course of the control error for the respective learning rate.

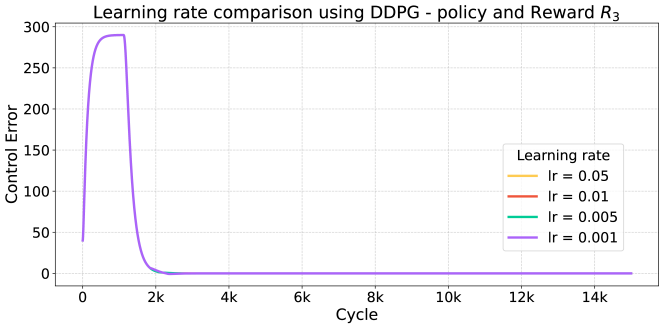

Fig. 6: Experiment 3: Variation of the learning rate when training with DDPG and R_3.

It clearly shows that the influence of the learning rate in the DDPG and R_3 combination has no impact, highlighting the robustness with respect to the learning rate.

VI. Conclusion and Future Work

In this paper, we presented a novel approach for a modular self-optimizing PID controller based on RL. It was designed to be used in both classical control scenarios and the Markovian decision process of RL. In both use cases, the reward function and the RL algorithm can be arbitrarily exchanged, significantly increasing the performance and flexibility for real-world applications.

The RLPID controller was implemented in version 2.1 of our open-source middleware MLPro, which we have specially extended to include the two sub-frameworks MLPro-BF-Control for classic and MLPro-OA-Control for online-adaptive control.

We demonstrated the performance of the RLPID controller in practical experiments with promising results. To ensure that the practical application of the new MLPro functionalities and all of our experiments can be reproduced, we have made our test program available in a public GitHub repository [1].

In future work, we will elaborate more on including disturbances and other variables to enhance the controller tuning. Also, simplified algorithms like bandit algorithms [12] or best response learning [28] will be considered in future work.

References

[1] Detlef Arend, Amerik Toni Singh Padda, and Andreas Schwung. "GitHub repository fhswf/paper-daieee-codit-2025". Feb. 2025. DOI: 10.5281/zenodo.14827652.

[2] Detlef Arend et al. "MLPro 1.0 - Standardized reinforcement learning and game theory in Python". In: *Machine Learning with Applications* 9 (2022), p. 100341. DOI: 10.1016/j.mlwa.2022.100341.

[3] Detlef Arend et al. "MLPro — An integrative middleware framework for standardized machine learning tasks in Python". en. In: *Software Impacts* 14 (Dec. 2022), p. 100421. DOI: 10.1016/j.simpa.2022.100421.

[4] Shalabh Bhatnagar et al. "Natural actor–critic algorithms". In: *Automatica* 45.11 (2009), pp. 2471–2482. DOI: https://doi.org/10.1016/j.automatica.2009.07.008.

[5] E.F. Camacho and C.B. Alba. "Model Predictive Control". Advanced Textbooks in Control and Signal Processing. Springer London, 2013. ISBN: 9780857293985.

[6] Myisha A. Chowdhury and Qiugang Lu. "A Novel Entropy-Maximizing TD3-based Reinforcement Learning for Automatic PID Tuning". In: *2023 American Control Conference (ACC)*. 2023, pp. 2763–2768.

[7] Oguzhan Dogru et al. "Reinforcement Learning Approach to Autonomous PID Tuning". In: *2022 American Control Conference (ACC)*. IEEE, June 2022, pp. 2691–2696. DOI: 10.23919/acc53348.2022.9867687.

[8] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". 2018. arXiv: 1801.01290 [cs.LG].

[9] EA Joseph and OO Olaiya. "Cohen-coon PID tuning method; A better option to Ziegler Nichols-PID tuning method". In: *Enginerring Research* 2.11 (2017), pp. 141–145.

[10] Ingenieurbüro Dr. Kahlert. "Download-Section: https://www.kahlert.com/download/". 2025.

[11] Ayub I. Lakhani, Myisha A. Chowdhury, and Qiugang Lu. "Stability-preserving automatic tuning of PID control with reinforcement learning". In: *Complex Engineering Systems* 2.1 (2022).

[12] Tor Lattimore and Csaba Szepesvári. "Bandit algorithms". Cambridge University Press, 2020.

[13] Nathan P. Lawrence et al. "Deep reinforcement learning with shallow controllers: An experimental application to PID tuning". In: *Control Engineering Practice* 121 (2022), p. 105046. DOI: https://doi.org/10.1016/j.conengprac.2021.105046.

[14] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". 2019. arXiv: 1509.02971 [cs.LG].

[15] "MLPro - Online documentation". https://mlpro.readthedocs.io. 2025.

[16] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". 2016. arXiv: 1602.01783 [cs.LG].

[17] Jose Pinheiro de Moura, João Viana da Fonseca Neto, and Patricia Helena Moraes Rego. "A neuro-fuzzy model for online optimal tuning of PID controllers in industrial system applications to the mining sector". In: *IEEE Transactions on Fuzzy Systems* 28.8 (2019), pp. 1864–1877.

[18] Rui Nian, Jinfeng Liu, and Biao Huang. "A review on reinforcement learning: Introduction and applications in industrial process control". In: *Computers & Chemical Engineering* 139 (2020), p. 106886.

[19] Katsuhiko Ogata. "Modern control engineering fifth edition". 2010.

[20] Sanjeev Kumar Pandey et al. "A robust auto-tuning scheme for PID controllers". In: *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*. IEEE. 2020, pp. 47–52.

[21] John Schulman et al. "Proximal Policy Optimization Algorithms". 2017. arXiv: 1707.06347 [cs.LG].

[22] Dorothea Schwung et al. "Self Learning in Flexible Manufacturing Units: A Reinforcement Learning Approach". In: *2018 International Conference on Intelligent Systems (IS)*. 2018, pp. 31–38. DOI: 10.1109/IS.2018.8710460.

[23] Dorothea Schwung et al. "Decentralized learning of energy optimal production policies using PLC-informed reinforcement learning". In: *Computers & Chemical Engineering* 152 (2021), p. 107382. DOI: https://doi.org/10.1016/j.compchemeng.2021.107382.

[24] T. Shuprajhaa, Shiva Kanth Sujit, and K. Srinivasan. "Reinforcement learning based adaptive PID controller design for control of linear/nonlinear unstable processes". In: *Applied Soft Computing* 128 (2022), p. 109450. DOI: https://doi.org/10.1016/j.asoc.2022.109450.

[25] Yuanping Su, Qiuming Yu, and Lu Zeng. "Parameter self-tuning pid control for greenhouse climate control problem". In: *IEEE Access* 8 (2020), pp. 186157–186171.

[26] Richard S Sutton and Andrew G Barto. "Reinforcement learning: An introduction". MIT press, 2018.

[27] Richard S Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999.

[28] Brian Swenson, Ryan Murray, and Soummya Kar. "On best-response dynamics in potential games". In: *SIAM Journal on Control and Optimization* 56.4 (2018), pp. 2734–2767.

[29] John G Ziegler and Nathaniel B Nichols. "Optimum settings for automatic controllers". In: *Transactions of the American society of mechanical engineers* 64.8 (1942), pp. 759–765.