UNIVERSITY OF
MARYLAND

Technical Documentation on Double Q-PID algorithm for mobile robot control

Highlights (link to Research Paper)

- A model-free reinforcement learning algorithm for adaptive low-level PID control.
- Incremental state and action spaces discretization for fast reinforcement learning.

AUTHORS:
Rahul Karanam, UID:118172507
Sumedh Reddy Koppula, UID:117386066

University of Maryland- College Park, 2021
Supervisor: Dr. Waseem A. Malik, Maryland Robotics Center
INSTITUTE FOR SYSTEMS RESEARCH

Contents

Chapter 1

Abstract

Several robotics applications which required automatic control have been successfully controlled using a classical PID (proportional-integral-derivative) controller. Most of the classical[9] and existing algorithms[5] use adaptive methods[8] to tune the gains without considering the operative conditions of the environment. Using these tuning algorithms give better results but with a trade-off with computation and resources. Using Reinforcement techniques such as Q-learning will help to find the best controllable variables considering the operative conditions of the environment. Incorporating Reinforcement learning techniques such as q-learning to update these values using agents has become popular due to faster convergence.
Although, using a q-learning based agent to update the low-level controllable gains will result in overestimation resulting in poor optimization. The core idea of this paper is to develop an expert-based system to explore the environment and update the gain values simultaneously. As the agent becomes more advanced in exploring the environment, it can be used to perform more complex actions. We propose a Double Q- PID algorithm to avoid the overestimation and simultaneously update the low-level controllable gains by using expert agents by learning different states and updating states based upon the actions. We demonstrate the results in simulations using Husky and Hector Quadrotor.

Chapter 2

Introduction

Mobile robots have become more commonplace in commercial and industrial settings. Hospitals have been using autonomous mobile robots to move materials for many years. Warehouses have installed mobile robotic systems to efficiently move materials from stocking shelves to order fulfillment zones. Mobile robots are also a major focus of current research and almost every major university has one or more labs that focus on mobile robot research. Mobile robots are also found in industrial, military, and security settings.
Mobile robots have the capability to move around in their environment and are not fixed to one physical location. Mobile robots can be "autonomous" which means they are capable of navigating an uncontrolled environment without the need for guidance.

Problem Statement
A big challenge for autonomous robot control strategies is that robots must operate in multiple and uncertain environments, an expert system point of view, many proposals were made for controlling mobile robots for direct applications in industrial environments. However, the real-time adaptability needed for this type of application is still an open issue for the majority of these proposals

2.1 PID

PID Control stands for Proportional-Integral-Derivative[1] feedback control and corresponds to one of the most commonly used controllers used in industry. Its success is based on its capacity to efficiently and robustly control a variety of processes and dynamic systems while having an extremely simple structure and intuitive tuning procedures.
Although not comparable in performance with modern control strategies, it is still the best starting point when one has to start designing the autopilot for an unmanned aircraft. In fact, most existing attitude control functionalities found in commercial autopilots or open-source developments, rely on some sort of PID Controls.

The PID Controller consists of the additive action of the Proportional, the Integral, and the Derivative component. Not all of them have to be present, therefore we also often employ P-controllers, PI-controllers, or PD-controllers. For the remainder of this text, we will describe the PID controller, while any other version can be derived by eliminating the relevant components.

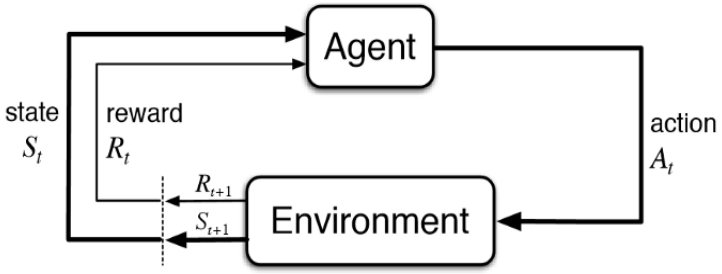
The PID controller bases its functionality on the computation of the "tracking error" e and its three gains K_p K_i, K_d. In their combination, they lead to the control action u, as shown in the following expression:

$$u(t) = K_{\mathrm{p}} \left(e(t) + \frac{1}{T_{\mathrm{i}}} \int_0^t e(\tau)\,\mathrm{d}\tau + T_{\mathrm{d}} \frac{\mathrm{d}e(t)}{\mathrm{d}t} \right)$$

2.2 Mathematical breakdown of our proposed algorithm

2.2.1 Reinforcement learning

The goal of Reinforcement Learning[7] is to learn a strategy for the agent from experimental trials and relatively simple feedback received. With the optimal strategy, the agent is capable of actively adapting to the environment to maximize future rewards.



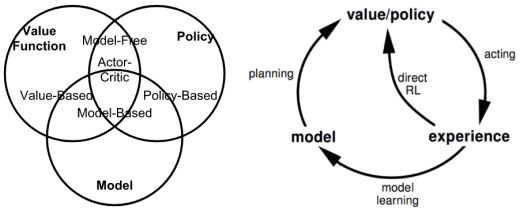
Block diagram of Reinforcement learning

1. Environment: Physical world in which the agent operates
2. State: Current situation of the agent
3. Agent: Feedback from the environment
4. Policy: Method to map agent's state to actions
5. Value: Future reward that an agent would receive by taking an action in a particular state

The agent is acting in feedback from the environment. How the environment reacts to certain actions is defined by a model, The usage of the model is optional. The agent can stay in one of many states ($s \epsilon S$) of the environment, and choose to take one of many actions ($a \epsilon A$) to switch from one state to another. The arrival of the agent in what state is decided by transition probabilities between states (P). Once an action is accomplished, the environment awards a reward ($r \epsilon R$) as feedback.
The reward function is defined in the model and transition probabilities. The model functionality might be known or unknown, hence we can discuss two of the below cases

1. **Case: Model based**: When planning is done with perfect information; we can use model-based RL. When we knew the environment, we can find the optimal solution by Dynamic Programming (DP).
2. **Case: Model free**: During learning with incomplete information; we do use model-free RL or we need to try to learn the model explicitly as part of the algorithm. Most of the following content serves the scenarios when the model is unknown.

The agent's policy (s) provides the guideline on what is the optimal action to take in a certain state with the goal to maximize the total rewards. Each state is associated with a value function V(s) predicting the expected amount of future rewards we are able to receive in this state by acting the corresponding policy. In other words, the value function quantifies how good a state is. Both policy and value functions are what we try to learn in reinforcement learning.



Various approaches in RL based on whether we want to model the value, policy, or the environment. (Image source: reproduced from David Silver's RL course lecture 1.)

The interaction between the agent and the environment involves a sequence of actions and observed rewards in time, t=1,2,. . . , T. During the process, the agent accumulates the knowledge about the environment, learns the optimal policy, and makes decisions on which action to take next so as to efficiently learn the best policy.

Let's label the state, action, and reward at time step t as St, At, and Rt, respectively. Thus the interaction sequence is fully described by one episode (also known as "trial" or "trajectory") and the sequence ends at the terminal state S_t

$$S_1, A_1, R_2, S_2, A_2, \ldots, S_t$$

Below are frequently used terminologies in understanding RL:

1. **Model-based:** Rely on the model of the environment; either the model is known or the algorithm learns it explicitly.
2. **Model-free:** No dependency on the model during learning.
3. **On-policy:** Use the deterministic outcomes or samples from the target policy to train the algorithm.
4. **Off-policy:** Training on the distribution of transitions or episodes produced by a different behavior policy rather than that produced by the target policy.
5. **Model: Transition and Reward:** The model is a descriptor of the environment. With the model, we can learn or infer how the environment would interact with and provide feedback to the agent. The model has two major parts, transition probability function P and reward function R

Let's say when we are in state s, we decide to take action a to arrive in the next state s' and obtain reward r. This process is known as one transition step, These transition steps are represented by a tuple (s, a, s', r).
The transition function $\mathbb{P}$ records the probability of transitioning from state s to s' after taking action a while obtaining reward r. We denote $\mathbb{P}$ for "probability".

$$\mathbb{P}(s, r|s, a) = \mathbb{P}[S_t + 1 = s, R_t + 1 = r|S_t = s, A_t = a] \tag{2.1}$$

Thus the state-transition function can be defined as a function of $\mathbb{P}(s, r|s, a)$:

$$P^a_{ss'} = \mathbb{P}(s^{'}|s, a) = \mathbb{P}[S_{t+1} = s^{'}|S_t = s, At = a] = \sum_{r \in R} \mathbb{P}(s^{'}, r|s, a) \tag{2.2}$$

The reward function R predicts the next reward triggered by one action:

$$\mathbb{R}(s,a) = \mathbb{E}[R_t + 1|\mathbb{S}_t = s, \mathbb{A}_t = a] = \sum_{r\in R} r \sum_{r\in R} P(s^{'}, r|s, a) \tag{2.3}$$

Policy
The policy is an agent's behavior function , which tells us which action to take in state s. It is a mapping from state s to action a and can be either deterministic or stochastic:

1. Deterministic: (s)=a
2. Stochastic: (a|s)=P[A=a|S=s]

Value Function
Value function measures the goodness of a state or how rewarding a state or an action is by a prediction of future reward. The future reward, also known as return, is a total sum of discounted rewards going forward. Let's compute the return Gt starting from time t:

$$\mathrm{G}_t = \mathrm{G}_{t+1} + \gamma \mathrm{G}_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k \mathrm{R}_{t+k+1} \tag{2.4}$$

The discounting factor $\gamma \in$ [0,1] penalize the rewards in the future, because:

1. The future rewards may have higher uncertainty; i.e. stock market.
2. The future rewards do not provide immediate benefits; i.e. As human beings, we might prefer to have fun today rather than 5 years later ;).
3. Discounting provides mathematical convenience; i.e., we don't need to track future steps forever to compute return. We don't need to worry about the infinite loops in the state transition graph.

The state-value of a state s is the expected to return if we are in this state at time t, $\mathrm{S}_t = s$:

$$\mathrm{V}(s) = E[\mathrm{G}_t|\mathrm{S}_t = s] \tag{2.5}$$

Similarly, we define the action-value ("Q-value"; Q as "Quality" I believe?) of a state-action pair as:

$$\mathrm{Q}(s,a) = \mathrm{E}[\mathrm{G}_t|\mathrm{S}_t = s, \mathrm{A}_t = a] \tag{2.6}$$

Additionally, since we follow the target policy , we can make use of the probability distribution over possible actions and the Q-values to recover the state-value:

$$V_\pi(s) = \sum_{a \in A} Q_\pi(s,a)\pi(a|s) \tag{2.7}$$

The difference between action-value and state-value is the action advantage function ("A-value"):

$$A\pi(s,a) = Q\pi(s,a)V\pi(s) \tag{2.8}$$

Optimal Value and Policy
The optimal value function produces the maximum return:

$$V_*(s) = max_\pi V_\pi(s), Q_*(s,a) = max_\pi Q_\pi(s,a)$$

The optimal policy achieves optimal value functions:

$$\pi_* = argmax_\pi V_\pi(s),_* = argmax_\pi Q_\pi(s,a)$$

we have

$$V\pi_*(s) = V_*(s)$$
$$Q_*(s,a) = Q_*(s,a)$$

2.2.2 Q Learning

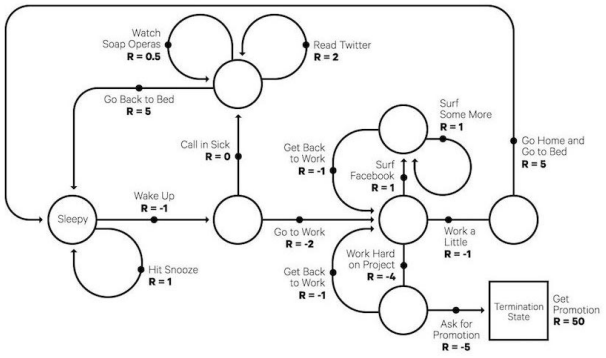
Generally RL problems can be viewed as Markov Decision Processes. All states in MDP has "Markov" property, referring to the fact that the future only depends on the current state, not the past:

$$\mathbb{P}[\mathrm{S}_{t+1}|\mathrm{S}_t] = \mathbb{P}[\mathrm{S}_{t+1}|S1, ..., \mathrm{S}_t]$$

It can be said that the future and the past are **conditionally independent** given the present, as the current state encapsulates all the statistics we need to decide the future.
A Markov decision process consists of five elements M = ⟨S,A,P,R,γ⟩:

1. **S** - a set of states
2. **A** - a set of actions
3. **P** - transition probability function
4. **R** - reward function
5. γ - discounting factor for future rewards. In an unknown environment, we do not have perfect knowledge about P.

A typical workday MDP example



Markov decision process

Bellman equations

It refers to a set of equations that decompose the value function into the immediate reward plus the discounted future values.

$$\begin{gathered} V(s) = E[\mathrm{G}_t|S_t{=}\mathrm{s}] \\ {=}E[\mathrm{R}_{t+1}{+}R_{t+2}{+}2R_{t+3}{+}\ldots|S_t{=}\mathrm{s}]) \\ {=}E[\mathrm{R}_{t+1}{+}(R_{t+2}{+}R_{t+3}{+}\ldots)|S_t{=}\mathrm{s}] \\ {=}E[\mathrm{R}_{t+1}{+}G_{t+1}|\mathrm{St}{=}\mathrm{s}] \\ {=}E[\mathrm{R}_{t+1}{+}\mathrm{V}(S_{t+1})|S_t{=}\mathrm{s}] \end{gathered}$$

Similarly for Q-value,

$$\begin{gathered} Q(s,a) = E[\mathrm{R}_{t+1} + \gamma V(\mathrm{S}_{t+1})\mathrm{S}_t = s, \mathrm{A}_t = a] \\ {=}\mathrm{E}[\mathrm{R}_{t+1} + E_{a\ \pi} Q(\mathrm{S}_{t+1}, a)\mathrm{S}_t = s, \mathrm{A}_t = a] \end{gathered}$$

2.2.3 Temporal-Difference Learning

Temporal-Difference (TD) Learning is model-free and learns from episodes of experience. However, TD learning can learn from incomplete episodes and hence we don't need to track the episode up to termination.

Bootstrapping

TD learning methods update targets with regard to existing estimates rather than exclusively relying on actual rewards and complete returns as in MC methods. This approach is known as bootstrapping.

Value Estimation

The key idea in TD learning is to update the value function V(St) towards an estimated return $\mathrm{R}_{t+1}{+}\gamma V(\mathrm{S}_{t+1})$ (known as "TD target"). To what extent we want to update the value function is controlled by the learning rate hyperparameter α:

$$\begin{gathered} V(\mathrm{S}_t) \leftarrow (1-\alpha)V(\mathrm{S}_t) + \gamma Gt \\ \mathrm{V}(\mathrm{S}_t) \leftarrow V(\mathrm{S}_t) + \alpha(\mathrm{G}_t V(\mathrm{S}_t)) \\ \mathrm{V}(\mathrm{S}_t) \leftarrow (\mathrm{S}_t) + \alpha(\mathrm{R}_{t+1} + \gamma V(\mathrm{S}_{t+1})V(\mathrm{S}_t)) \end{gathered}$$

Similarly, for action-value estimation:

$$Q(\mathrm{S}_t), \mathrm{A}_t)) \leftarrow Q(\mathrm{S}_t), \mathrm{A}_t)) + \alpha(\mathrm{R}_{t+1}) + \gamma Q(\mathrm{S}_{t+1}), \mathrm{S}_{t+1}))Q(\mathrm{S}_t), \mathrm{A}_t)))$$

SARSA: On-Policy TD control

"SARSA" refers to the procedure of updating Q-value by following a sequence of ..., $\mathrm{S}_t, \mathrm{A}_t, \mathrm{R}_{t+1}, \mathrm{S}_{t+1}, \mathrm{A}_{t+1}$ The idea follows the same route of GPI. Within one episode, it works as follows:

1. Initialize t=0

2. Start with S0 S 0 and choose action $A_0 = argmax_{a \in A} Q(S_0, a) where \in$-greedy is commonly applied

3. At time t, after applying action A_t we observe reward R_{t+1} and get into the next state S_{t+1}

4. Then pick the next action in the same way as in step 2: $A_{t+1} = argmax_{a \in A} S_{t+1}(St + 1, a)$

5. Update the Q-value function:
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_t + 1 + \gamma Q(S_{t+1}, A_{t+1}) Q(S_t, A_t))$

6. Set t=t+1 and repeat from step 3.

In each step of SARSA, we need to choose the next action according to the current policy.

2.2.4 Off-policy TD control

Algorithm for Q-Learning: Off-policy TD control

1. Initialize t=0

2. Starts with S_0

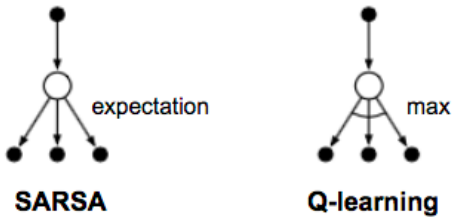
3. At time step t , we pick the action according to Q values, $A_t = argmax_{a \in A} Q(S_t, a) \in$ -greedy is commonly applied.

4. After applying action At A t , we observe reward R_{t+1} and get into the next state S_{t+1}

5. Update the Q-value function: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Max_{a \in A} Q(S_{t+1}, a) Q(S_t, A_t))$

6. t=t+1 and repeat from step 3.

The key difference from SARSA is that Q-learning does not follow the current policy to pick the second action $A_t + 1$. It estimates Q_* out of the best Q values, but which action (denoted as a star) leads to this maximal Q does not matter and in the next step, Q-learning may not follow a star.



Comparison of Q learning and SARSA

2.3 Reinforcement learning for Adaptive PID using Q-Learning

From the definition of the classical PID control:

$$u(kT) = u(kTT) + k1e(kT) + k2e(kTT) + k3e(kT2T) \tag{2.9}$$

where,

$$k_1 = k_p(1 + k_d/T) \tag{2.10}$$

$$k_2 = k_p(1 + 2(k_d/T)T/k_i) \tag{2.11}$$

$$k_3 = k_p(kd/ki) \tag{2.12}$$

being k_p, k_d, k_i are the gains of the PID controller, T is the sampling time and e (k T) the error term used to form the derivative, proportional and integral terms that in turn yield the control action u (k T) at time instant k T , with k = 0 , 1 , 2 , . .
By arranging parameters of PID controllers in an action vector
$k = (k^1, k_2, ..., k_D)^T$ where D is the number of PID controllers. So that an RL agent needs to select the action vector in order to achieve a certain goal.
During this process agents learn in feedback loop from the environment and fixing the PIDs gains $k^1 = (k_p^1, k_i^1, k_d^1), ..., k^D = (k_p^D, k_i^D, k_d^D)$ through the learned control policy .
As described in Chapter 1, The decision problem in RL can be formulated as a Markov Decision Process. Where, agents find an optimal policy * to define optimal controller parameters() for different system states() such that the total expected reward is maximized by:

$$J^* = MaxJ_\pi = max\mathrm{E}_\pi \mathrm{R}_t | x_t = x \tag{2.13}$$

optimal state-action value function can be defined as

$$Q^*(x_t, k_t) = E\left\{r_{t+1} + \gamma \cdot maxQ^*(x_{t+1,k_{t+1}}) | x_t, k_t)\right\} \tag{2.14}$$

here,

1. r_{t+1} indicates instantaneous reward acheived by the system trabsition from the state x_t to x_{t+1}
2. γ indicates discount rate

Depending on the optimal state-action function learning rate, the optimal policy is obtained as:

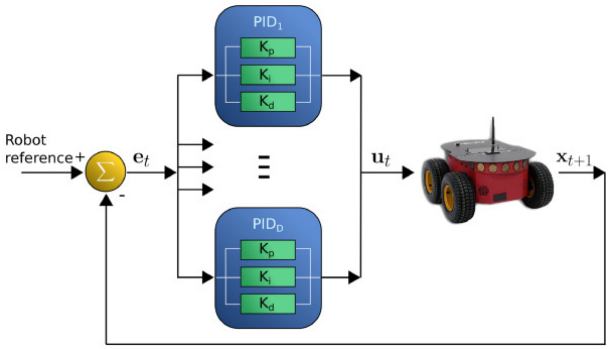
$$\pi^* = argmax_k Q^*(x_t, k) \tag{2.15}$$

As discussed in the topic 2.3.4, the optimal policy π^* is solved through off-policy model-free Q learning algorithm:

$$Q(x_t, k_t) \leftarrow Q(x_t, k_t) + \alpha(R_{t+1} + \gamma Max_k Q(x_{t+1}, k_{t+1}) - Q(x_t, k_t)) \tag{2.16}$$

2.4 Double Q-Learning

In order to reduce the over-estimation present in the original Q -learning algorithm, Double Q-Learning algorithm is introduced; This is achieved by a double estimator for the Q function, Q_A, Q_B



From the above equations, each Q-Learning has its own off-policy model-free algorithms. Hence, each Q function should update among themselves in opposite
When policy uses the function Q_A to choose the action, Then the updating rule is given as follows

$$Q_A(x_t, k_t) \leftarrow Q_A(x_t, k_t) + \alpha(R_{t+1} + \gamma Max_k Q_A(x_{t+1}, k_{t+1}) - Q_A(x_t, k_t)) \tag{2.17}$$

Similarly, Updating rule for Q_B is give as follows

$$Q_B(x_t, k_t) \leftarrow Q_B(x_t, k_t) + \alpha(R_{t+1} + \gamma Max_k Q_B(x_{t+1}, k_{t+1}) - Q_B(x_t, k_t)) \tag{2.18}$$

In this way we can overcome the bias in the value estimation.

Chapter 3

Double Q-PID algorithm and its Implementation

The algorithm used in this paper is the Double Q-PID DQPID[4] algorithm which is an incremental model-free formulating an artificial expert agent to predict and update the low-level PID gains for the PID controllers in a real-time environment.

During the evolution of Reinforcement learning many algorithms have come into existence, but Q-Learning was popular amongst all others as it has a wide range of applications in fields of Economics, Games, Controllers, and Optimization related action-state dynamic systems. Later in the 2000s, researchers found flaws with Q-learning algorithms of overestimating the rewards based on the initial states.

In order to give a proper estimation of the rewards and improve the performance of the model without overshooting the gains or rewards, researchers developed double Q-learning algorithms which have resolved the overestimation problem by layering two q-functions to estimate accurately by considering the overall rewards rather than depending upon intermediate or starting state.

This algorithm has increased the performance dramatically on calculating the action rewards of stochastic processes. We use this algorithm to find the optimal non-uniform state and update our PID low-level features of our model in real-time using the incremental active learning exploration and exploitation technique. We will discuss each mechanism in much more detail in the next sections:

3.1 Incremental Learning

In general, in an RL problem, the state space is represented through a finite set of state-action spaces or through uniform discretization. This method will make the model reach different action-state pairs which is resulting in higher computation complexity, bias caused due by optimization errors when running this algorithm in real-time. In order to avoid all these problems explained earlier, this paper presents incremental discretization of its state and action spaces.

Incremental active learning[2] is the process of incrementally discretizing the state and action space into finer dimensions where these discretized versions will only constitute promising features of the possible solution area. It decreases the need of exploring a large area by

decreasing the concentration of exploring a relevant space and focusing on it. This way of learning will reduce the overall computational cost and increase the latency between the states by focussing on the possible states. We can relieve the curse of higher dimensionality in large datasets by using this learning.

First, let me define all the parameters used in this section.

$$\mathbf{X} = \{x_i, ..., x_{n+1}\} \tag{3.1}$$

$$\eta(x_{t+1}, x_t) > \rho \tag{3.2}$$

Membership Function $= \eta(x_t, x_{t+1})$

This function is used to check whether the new transformed state x_{t+1} is in the vicinity by calculating the euclidean distance between them.

$$\eta(\mathbf{x_t}, \mathbf{x_{t+1}}) = \sqrt{\sum_{i=1}^{n}(x_t - x_{t+1})^2} \tag{3.3}$$

Threshold value - ρ
This is a scalar value to represent the threshold value for checking whether to include x_{t+1} is in the vicinity to x_t

x_t - current system state at time -t

k_t - vector of controllable gain parameters

X is our state-space representation defining our states and U is our action space representation which is a low-level control that will cause the states x_i to x_{i+1}. Assume that our initial state is in state x_0 and a u_0 which is a low-level control is being applied to the x_0 state to change the system to a new state x_1.

First, we need to check whether the state member x_i is already a member of our known state-space representation i.e X. If this state is not in the known state-space then-new state x_{i+1} is added to the set of state-space representation, whereas if the new state x_{i+1} is already in the state space then we don't update the state (X). This procedure is sequentially repeated whenever a controller and the robot agent interact based upon the controllable variables given to it.

We use a membership function equation 3.2 for every transition state to check whether the new transition state is providing any useful information in comparison to the previous state i.e if the equation 3.3 is satisfied, then only x_{i+1} is added to the known state $X = X \cup x_{i+1}$.

We use a threshold ρ which calculates the euclidean distance between both the pairs equation $\eta(\mathbf{x_t}, \mathbf{x_{t+1}})$, and use this criterion to decide whether to include this new state in our known state-space or not. If the new state is closer to the previous state then the Q values are updated, if it is far from the neighboring state or outside the boundary of the known state, then it is incorporated into the known state space.

$$\wedge = (\rho_1, \rho_2, ..., \rho_z) \tag{3.4}$$

ρ_i - discretization step for state space

$\wedge$ - Discretization space for steps until ρ_z i.e maximum level

$$\Delta = (\delta_1, \delta_2, ..., \delta_j) \tag{3.5}$$

δ_i - discretization step for action space

Δ - Discretization space for steps until δ_j i.e maximum level

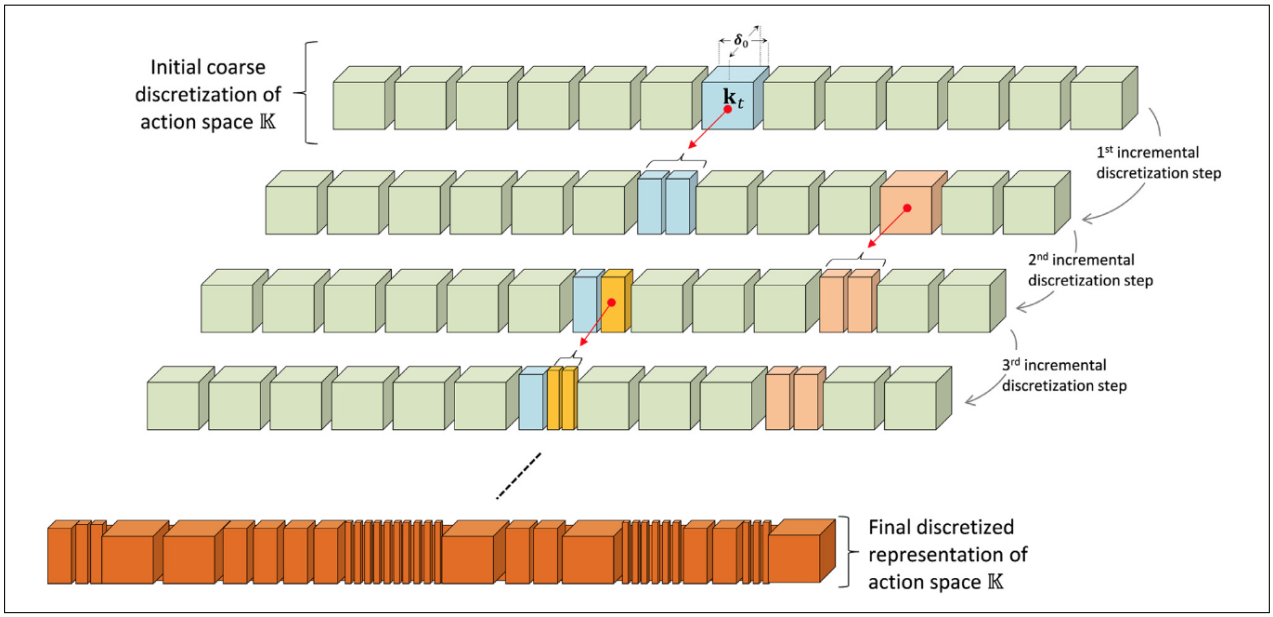


Figure 3.1: Incremental discretization of action space.

Let me give you an example of this process below for the incremental discretization of action space:
Considering initial action space:

1. First the states and action spaces are initialized uniformly or coarsely.
2. Suppose the system is at a time t, state x_t and agent chooses an action k_t from the given action state K using the greedy approach given by 2.15 from the control policy pi.
3. Now, an action kt has applied on the x_t which will transfer to the next state x_{t+1}, the observed state(x_{t+1}) is checked using the membership function η by calculating the euclidean distance between x_{t+1} and x_t i.e is ρ, if that function is greater than ρ then it is included in our state-space representation, else if it is in the vicinity of the previous state, then the Q values are updated based upon the transition state.
4. If the system remains invariant to these changes then, the system will choose the same action over and over and the action space will be more discretized in certain regions where the possible or the best actions are suggested.
5. Then, after N successive states, verifying with the successor state to the previous state, if the agent still chooses the same action from the K then the state space configuration will be finite offering a higher discretized action state.

6. This procedure is repeated for every discretized step δ_z that the system is invariant to those actions and the K will be evolved to a final discretized state representing the action space K.

7. This process of discretizing the action space until it reaches a maximum level denoted by J(Δ) or δ_z which is known as the discretized step for each discretization level of the state space representation.

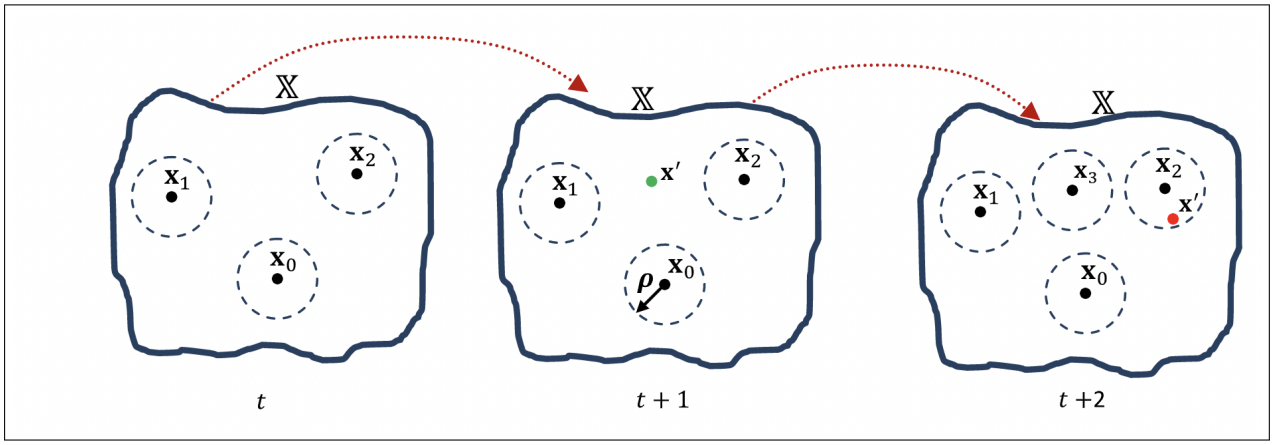


Figure 3.2: Incremental discretization of state space.

Furthermore, we need to repeat a similar process for finding the incremental discretized version of the state space. Refer [3.2] for an example of discretization of state space using incremental active learning.

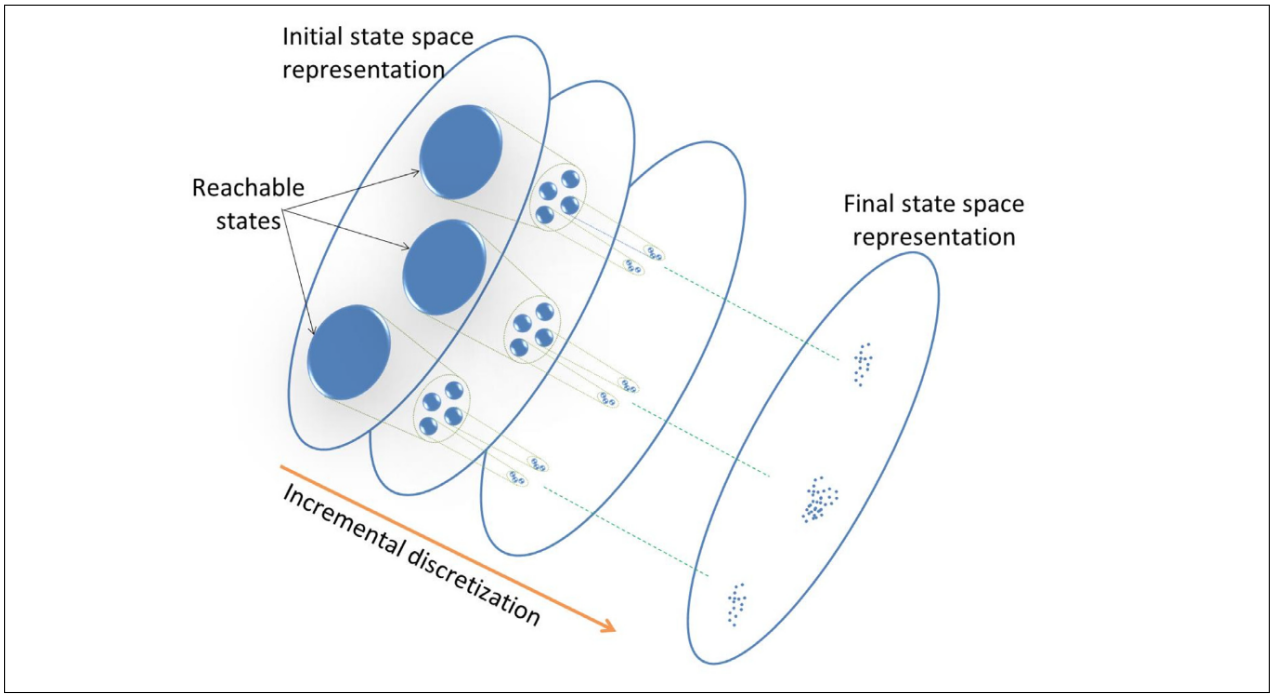


Figure 3.3: Represntation of incremental active learning

To show how the state space discretization works please refer to the [3.3]. Now, consider we have a initial state x_t at time t , as we transit to the next state after applying an action kt we move into a new state x_{t+1} , as before we have explained in the previous discretization of action space we check the observed statex_{t+1} using the membership function to calculate

the euclidean distance between the x_{t+1} and x_t if $\eta(x_t, x_{t+1}) > \rho$ then it is incorporated in the state space representation.For every time step, we check for the membership function and try to refine our space and discretize our state space into favorable spaces.

3.1.1 Temporal Memory

We use temporal memory in order to support or govern the incremental discretization process of state and action space. Now we define $Mt = (xt, kt)$ as the temporal memory at a time t, which corresponds to the information about the state and action space. This memory is used to selectively learn the promising state and action spaces as per the system behavior and well reducing the total computational load by focusing on the limited subspaces of learning.

$$M_t = (x_t, k_t) \tag{3.6}$$

x_t - State at a time t
k_t - action at a time t

We define temporal memory as a storage buffer which is defined as a tuple containing the current state and action at time step t. For every episode, we update the temporal memory for every new state i.e after interacting with the agent, we need to check whether the system is invariant or variant to these interactions.

We determine whether a system is variant or invariant by comparing the current temporal state with the last N temporal memories i.e $M_t = M_{t-1} = \ldots M_{t-N}$. If it is not equal, then the system is variant to these changes; Therefore, we need to incorporate those new transitions into our replay buffer for later training purposes.

Conversely, if a system is invariant to these changes then we can replay our experience replay buffer to update the Q values from equations 2.17 2.18 function Q_A and Q_B . If the system is invariant to these changes that implies that we need our model to explore more state and action spaces to reach better states in the future. We do that using the incremental discretization of action and state space to get new states and activities to look for.

3.2 Stochastic Experience Replay

The paper states about experience replay as a memory buffer in order to store the previous states and use it later as a backup to reuse these past states for better understanding of the environment and use these experiences to increase our performance. It is also known as Prioritized experience replay (PER) stated in [6] saves transitions of the experiences with a certain size limit for the buffer.

These are later used to train new agents for training. This would help the agent to understand and have a better prediction or decision-making on that state. Two factors mainly govern this buffer. First, we should know how many experiences to retain in that buffer and the latter is how to sample that is to be replayed in the buffer for training purposes. The overall stability and performance of a model are greatly dependent on how well we use the buffer to replay the past states or experiences.

Experience replay is used here in order to update the low-level control gains of the PID model in real-time.

The technique used in this paper is taken from [10] which explains the mechanism of storing sample transitions to the buffer and constantly updating the buffer with recent experiences.

R-Buffer Size It describes the maximum number of transition states to be stored in the buffer.This buffer size can store up to a maximum of **'m'** states $R = \tau_1, \tau_2, ...\tau_m$.

$$\tau_i = (x_t, u_t, r_t, x_{t+1}). \tag{3.7}$$

A transition is defined as a quadruple consisting of four parameters. τ_i- transition
x_t - state
k_t - action
x_{t+1}-transition of next state from x_t by applying the action k_t to x_{t+1}
r_t- reward received after performing the action a to move from x_t to x_{t+1}.

Stochastic experience replay stores the latest transition state from the given real transitions (τ).First, we initialize the stochastic replay buffer along with the value functions Q_a and Q_b.

The key idea of stochastic experience replay is to train the agent from the buffer of previously recorded experienced transitions. This recording is done on the basis of whether the system is variant to the new transition state or not, if it is variant, then that current state transition experience is stored in the buffer. The state of this system is checked by comparing the previous memories of the Temporal memory M with the current transition. Using this replay mechanism has always had a trade-off with the buffer size as the larger the size the better the model but it trades off with the computational efficiency.

When recording the past state transition states reach a buffer size R, we try to remove the old experience buffer states to incorporate new experiences which will optimize the performance of the agent. At every iteration, the current state transition is added to the replay buffer and some of the stored transitions are replayed to the model to update the Q_a and Q_b value functions.

Unlike the conventional RL algorithms, where we need to set a sampling probability for playback of the recorded experiences, this paper uses random sampling of the experiences and updating the Q-values episodically. The sampling of the transition state from the buffer is chosen randomly for every time step. The main advantage of randomized updating the value function is that it will increase the reach to other unknown states which will contribute to more exploration.

It also reduces the overestimation of data i.e reducing the bias caused by the replaying correlated data if we had replayed and updated the q-values in an order. This majorly reduces the burden of choosing the better hyperparameters which minimizes or maximizes the effect on the algorithm.

3.3 DQ-PID pseudo algorithm

This section will explain the pseudo-code of the proposed DQPID algorithm.

Algorithm 1 Pseudo-code for the Double Q-PID algorithm.

1: **Inputs**: α, γ, $r(\cdot,\cdot)$, ϵ_0, $\mathbb{K}$, $\eta(\cdot,\cdot)$, n, m, N, Λ, Δ
2: $\mathbf{x}_0 \leftarrow$ get sensor readings
3: Initialize Replay Buffer R
4: Initialize $\mathbb{X}$
5: Initialize Q^A, Q^B
6: **loop**
7: ϵ - *greedy* action selection $\rightarrow \mathbf{k}_t$
8: Set the current temporal memory as $M_t = (\mathbf{x}_t, \mathbf{k}_t)$
9: Interact with the system, observe the transition state $\mathbf{x}_{t+1}$ and the obtained reward r_{t+1}
10: **if** the system is variant **then**
11: Store the current transition $\tau = (\mathbf{x}_t, \mathbf{k}_t, \mathbf{x}_{t+1}, r_{t+1})$ in the replay buffer R
12: Identify the nearest state $\mathbf{x}_i \in \mathbb{X}$ to $\mathbf{x}_{t+1}$
13: **if** $\mathbf{x}_{t+1}$ is in the vicinity of $\mathbf{x}_i$ **then**
14: Double Q-learning updating $\rightarrow$ Update Q^A and Q^B according to Eqs. (9) and (10)
15: **else**
16: Incorporate $\mathbf{x}_{t+1}$ in $\mathbb{X}$
17: Set up Q^A and Q^B for incorporating the new point $\mathbf{x}_{t+1}$
18: **end if**
19: **else**
20: Incremental active learning
21: **end if**
22: Stochastic experience replay
23: Set the system state for the new execution loop
24: **end loop**

Figure 3.4: Double Q-PID pseudo algorithm

Line-1 shows the input parameters which are related to the Double Q learning algorithm value functions and policy functions. These are defined below and you can refer to these parameters in the above set of sections for more understanding.

The proposed algorithm has input parameters that are commonly used in any reinforcement learning techniques (Q-learning algorithm) such as:

α - learning rate for the double q learning from 2.16
γ- discount rate 2.17
$r\,(.,.)$- reward function 4.3
ϵ - initial exploration probability

The below input parameters are closely related to the algorithm as these explain our proposed algorithm parameters.

K - initial set of coarse possible action spaces
$\eta(.,.)$ - Membership function 3.2 used in the incremental discretization process of action and state space.
n - Number of initial transitions or interactions made by the agent to initialize the replay buffer or the size.

m - Maximum size for the state transitions in $\mathbf{R} - \tau_1, \tau_2, ..., \tau_m$
N - It is the number of temporal memories needed to check whether the system is invariant or not.
$\wedge$ - Set of state discretization steps
Δ - Set of action discretization steps

From line-2 to line 5, we initialize the input variables required by the robot i.e the input x_0 is given by the sensor readings taken from the robot. After initializing the input state, we initialize the experience replay buffer R with a possible size of n. Later, we initialize our set of state-space variables **X** 3.2 containing all the initial states till n+1 which are interacted during the initialization of our input state x_0.

From lines 3 - 5, variables are initialized and are used to randomly update the state action values to the Q^a Q^b functions 2.17 and 2.18.

Once the action k_t is selected from the action states we initialize the temporal memory $M_t = (x_t, k_t)$ on line 8 with the current state and action from the previous initialization.After the selected action k_t , we try to update the gains of our PID low-level controllable variables with the selected action 2.9. After the interactions with the agent with the updated controllable variables, we observe the new state x_{t+1} and compute the reward r_{t+1} using the reward function 4.3.

Now we check whether the system is variant or invariant to these interactions above. First, we check the current transition state by comparing the previous N memories from the previous temporal memory $M_t = M_1 = ...M_N$

If the time t >= N then the system is variant to those changes, where we store the current transition i.e $\tau = (x_t, k_t, x_{t+1}, r_{t+1}$ to the replay buffer **R** to use it later for training. After storing the τ in the buffer, we check whether the new state x_{t+1} is in the vicinity of the previous state x_t using the membership function η 3.3. If $\eta(x_t, x_{t+1}) <= \rho$ then it is considered to be in the vicinity of of the previous state x_t, we update the Q-learning value functions Q^a Q^b given by 2.17 and 2.18.

On the other hand, if $\eta(x_t, x_{t+1}) > \rho$ i.e the new state x_{t+1} is not in the vicinity of the previous state x_t , we add this successor state x_{t+1} and incorporate to our set of state space **X** 3.2. After updating the new state , we further update the Q^a Q^b to incorporate the new state x_{t+1}.

Contrary, when t < N or the system is invariant to these interactions, then we try to update our state and action space representation by using our incremental discretization process mentioned in 3.4,3.5.

We further discretize the action and state spaces to get promising states and actions which will be again checked for every episode or step. After the discretization, we update the Q^a Q^b values by replaying the memories in the experience replay buffer using the stochastic replay mechanism 3.7. Now the system state is updated with the new state and again the loop will start where we again check if the system is invariant or not and later update the state and action spaces depending on the new state.

Chapter 4

Experimental setup

4.1 Experimental platforms

After analyzing the implementation of the double Q PID and its algorithm, In this technical document, we represent experimental simulation analysis of double QPID on 2 different robots. The robots used for experiments are

1. Unmanned Ground vehicle: Clear path's Husky- Mobile terrestrial robot

2. Unmanned Aerial vehicle: Hector Quadrotor

Due to lack of research equipment's and facilities we are not implementing our proposed algorithm on Underwater Robot and Pioneer Robot.

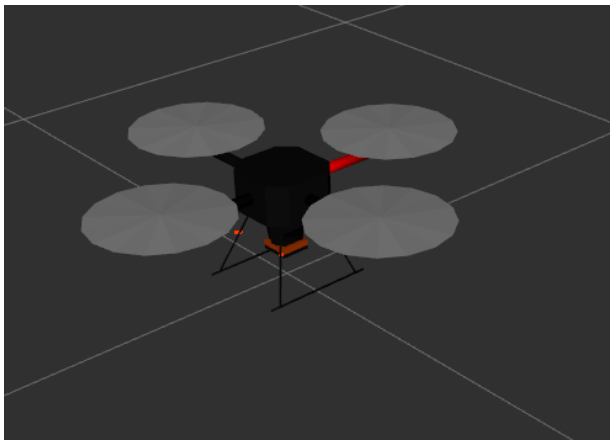
Unmanned Ground vehicle: Husky

Husky is a medium sized robotic development platform. Its large payload capacity and power systems accommodate an extensive variety of payloads, customized to meet research needs. Stereo cameras, LIDAR, GPS, IMUs, manipulators and more can be added to the UGV. Husky is fully supported in ROS with community driven Open Source code.

Unmanned Ground vehicle: Husky

Unmanned Aerial vehicle: Hector Quadrotor

Hector-quadrotor is a small flying robot which is fully supported in ROS and can be simulated in gazebo. This open source UAV is developed and maintained by Technische Universität Darmstadt.



Unmanned Aerial vehicle: Hector Quadrotor

We chose the above platforms to analyze the adaptive performance of the proposed algorithm in different dynamics and conditions.

4.1.1 Outcomes of the experimental setup:

1. Test the performance of the proposed double-QPID algorithms on the above-discussed platforms
2. Present and discuss the setup for the reinforcement learning experiments discussed in the research paper
3. Simulation results of the algorithm proposed

Reinforcement Learning Setup

The authors of the research paper had conducted series of experiments by using gains of the PID controller as the incremental Q-learning agents. They have employed one PID controller for each degree of freedom, thus:

$$k_t = (k_p^1, k_i^1, k_d^1, k_p^2, k_i^2, k_d^2, ..., k_p^D, k_i^D, k_d^D) \tag{4.1}$$

Here the value of D varies with application and the number of degrees of freedom.

4.1.2 RL setup for Husky AGV and Hector Quadrotor

Researchers considered the control movements along the horizontal plane so that they can control linear and angular speed only. Since we are controlling linear and angular speeds, we only have two degrees of freedom, $D = 2$.
For these both experimental platforms, The gains vector is represented by k_t and control action vector which has to be executed on the robot is represented as

$$u_t = (u_1, u_2) \tag{4.2}$$

Where,

1. u_1 is the control command for linear instantaneous speed

2. u_2 is the control command for the rotational instantaneous speed with respect to the vertical z axis

NOTE: The initial values of k_t are taken from a uniform random distribution such that $k^i_{min} \leq ki \leq k^i_{max}$. Authors randomly initialized the values of k_t to make sure the algorithm explores, Simultaneously, we make sure that the number of hyper-parameters that the user needs to supply is reduced. This helps in simplifying the user requirements when working with real-time platforms.

Reward function: The reward function is defined based on the operative requirements of the system. So, For our experimental platforms these operative requirements are velocities setpoints. A Gaussian function is used to shape the reward of the form:

$$r_t(x_t, x_{req}) = \frac{1}{2\pi\sigma} \exp(\frac{-||x_t - x_{req}||^2}{2\sigma^2}) \tag{4.3}$$

- σ is a free parameter that determines the width of the Gaussian function.
- $(||\cdot||)$ is the euclidean distance between the current state x_t and the set point x_{req} .
The current state, x_t, and the requested state,x_{req} , are vectors with dimensions varying depending on the degrees of freedom and the reward is a scalar value.
We have noticed that the Gaussian function always gives a high positive reward signal when the current system state (x_t) is close to its fixed target (x_{req}) and a lower reward is obtained when the distance between x_t and x_{req} is higher.
Exploration/Exploitation setup: This setup helps us to understand how the exploration-exploitation of the agent knowledge is implemented.

1. If the learned representation is exploited too early, the agent performance may reach a local maximum failing to see better policies due to the lack of exploration of the state-space

2. if the exploration is too high, the learned policy is never implemented since the agents randomize its behavior.

From the explanation, It is better to consider a value with an average threshold with a high exploration policy during the initial stages of learning, when the agent has very little knowledge of the problem, and over time change the behavior to that of exploiting the learned representation, something that commonly happens on later stages of the learning processes. A greedy exploration scheme with an exponential decay rate of the parameter t that represents the probability of the agent of taking an action based on the current policy, or rather take an exploratory action based on a random policy. Then, the decay rate is defined as:

$$\epsilon_t = \epsilon_0 + \epsilon_1 \exp^{-t}$$

For both experimental platforms we have considered $\epsilon_0 = 0.02$ and $\epsilon_1 = 0.3$.

Chapter 5

Results

5.1 Husky Robot simulation results

We have simulated the proposed algorithm on a AGV- mobile robot, where simulations experiments were performed using the Gazebo simulator and ROS. The two manipulated variables $u_t = (u_1, u_2)$ and the system state is defined as being $x_t = (v_x, w_z)$ here v_x and w_z are the controlled variables, i.e. the linear and angular velocity respectively. From our experimental setup of husky, there are two low-level controllers (D=2). We have assumed that, the initial values of the gains in the vector k_t are selected randomly in such a way that $0 \leq k_j^d \leq 2$
We have taken controlled variable as

$v_{xref} = 0.33m/s$
$w_{zref} = -1.9m/s$, shown in the Figure 5.1

Observations

1. From the Figure 5.1(right) it is evident that reaching desired velocity setpoint is achieved within seconds.

2. Figure 5.2(left) shows selected controller gains, after 100 seconds, the agent finds a suitable combination of controller gains by looping around.

3. Figure 5.2(right)incremental discretization of the action space evolves over time, reaching finally the fixed maximum level $l_{max} = 8$.

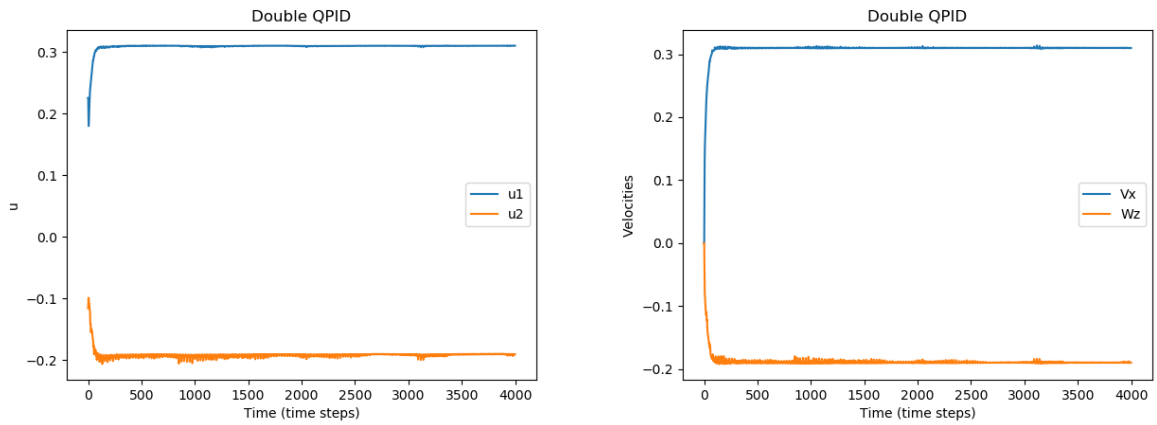


Figure 5.1: Controlled variables

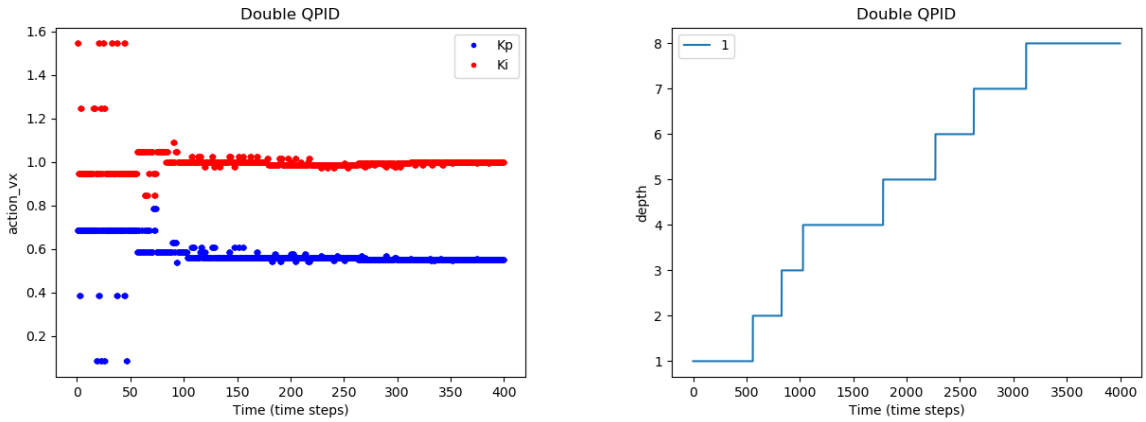


Figure 5.2: Gains for the low-level controllers with Discretization levels of the action space

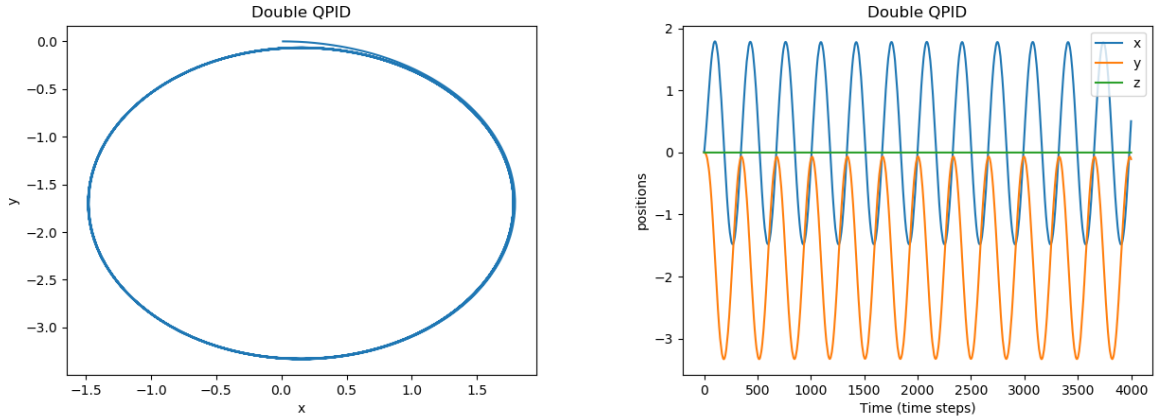


Figure 5.3: Pose and Positions

Explanation

1. According to Figure 5.2(right), after 100 seconds the agent is finding a set of gains to successfully drive the robot according to its operative desired setpoint.
   As a result, the action selection has stabilized, allowing the incremental discretization of the state and action spaces to take place by means of incremental active learning.
   As such, we can assume that the agent efficiently adapts to the required conditions and therefore specializes in the search for actions (controller gains) within a particular area.

2. Figure 5.3, Shows the robots posing while the action space evolves over time

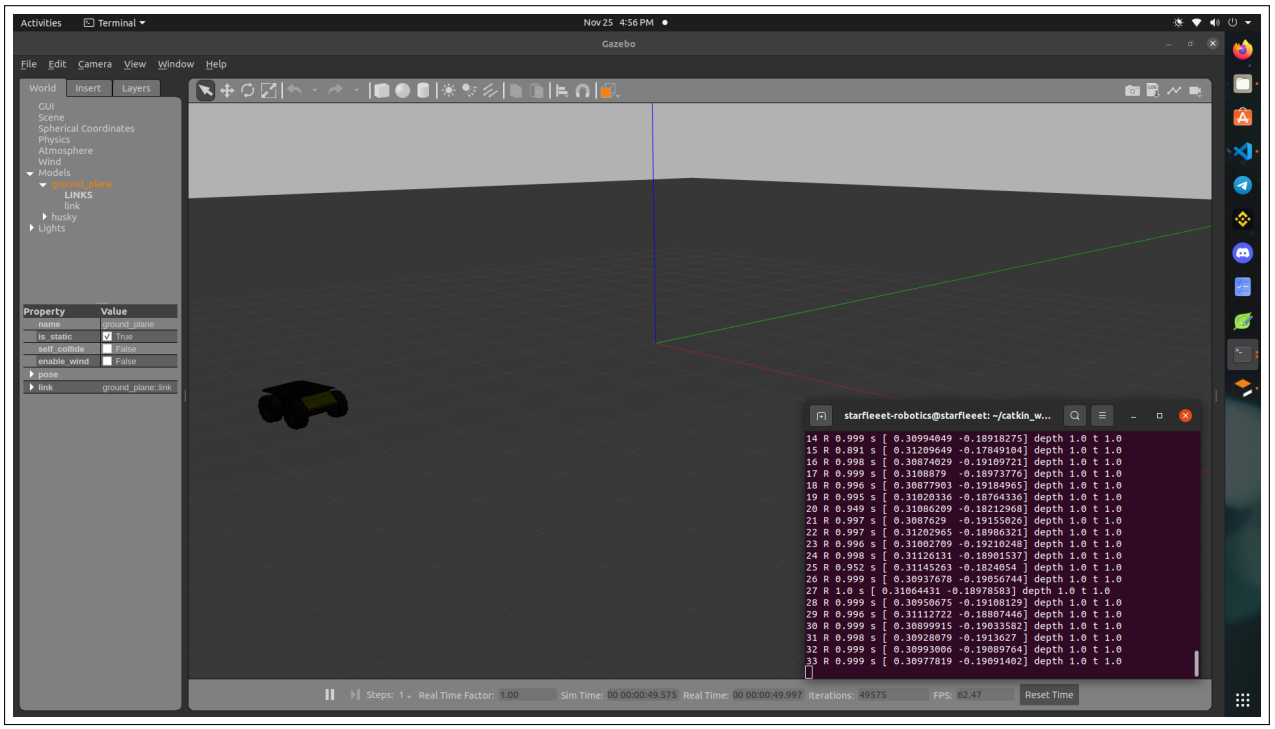


Figure 5.4: Gazebo simulation of Double Q-PID controlled Husky Robot

5.2 Hector Quadrotor simulation Results

Similar to that of Husky experiment, But We intent to control the flying robot in a horizontal plane. For this particular control task, once the robot is hovering, our proposed algorithm starts run with the following velocity request
$v_{xref} = 0.07m/s$'
$w_{zref} = 0.22m/s$

Observations

1. From the Figure 5.4, The linear and turning velocities reach their reference values after an early overshoot (around the first five seconds). However, the Double Q-PID algorithm rapidly stabilizes the controlled magnitudes.

2. Fig. 5.5 shows that, as the learning process progresses, the agent focuses on improving its policy, which leads to fewer actions being executed in the action space, corresponding to a higher discretization level of the state and action spaces.

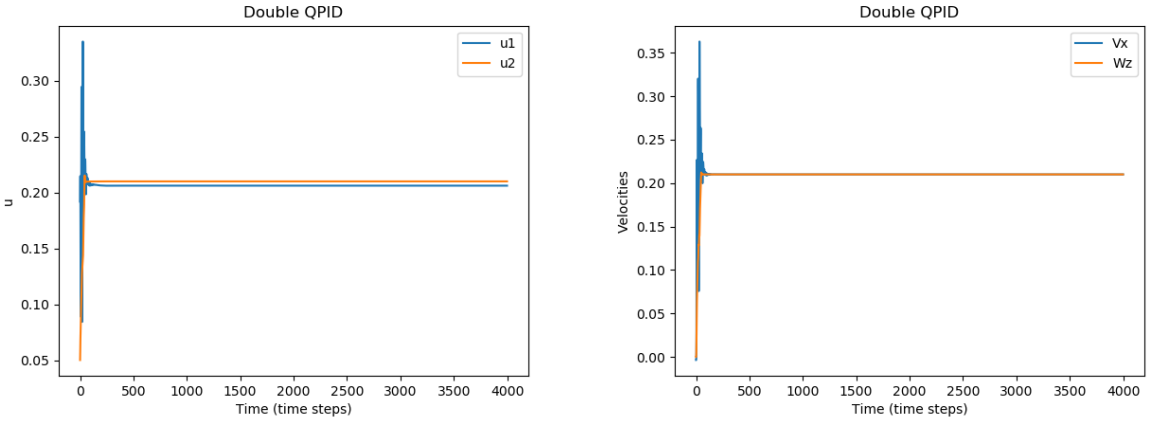


Figure 5.5: Controlled variables for Drone setup

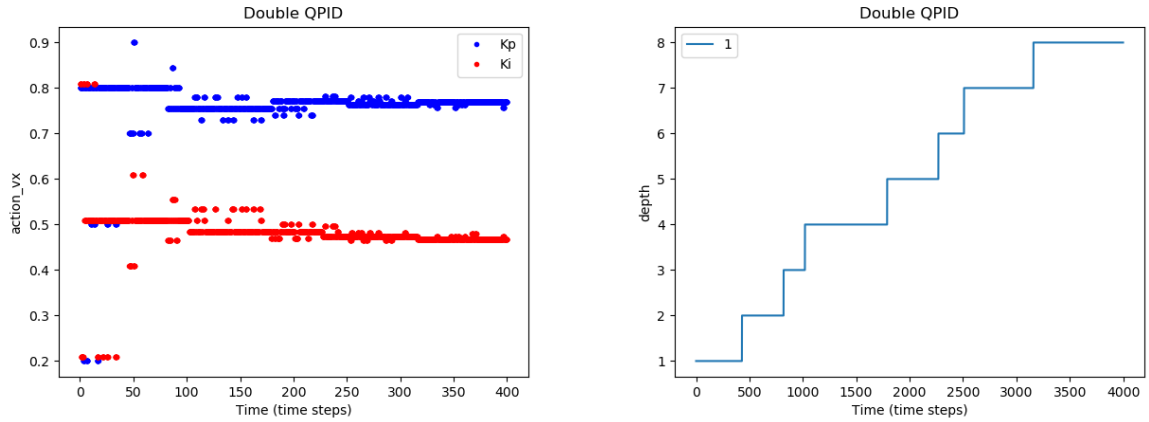


Figure 5.6: Gains for the low-level controllers with Discretization levels of the action space

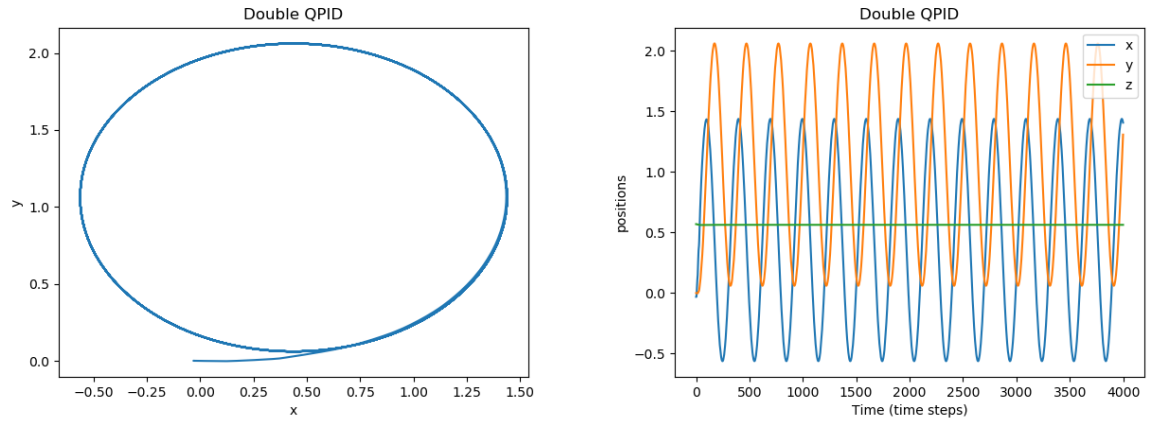


Figure 5.7: Pose and Positions

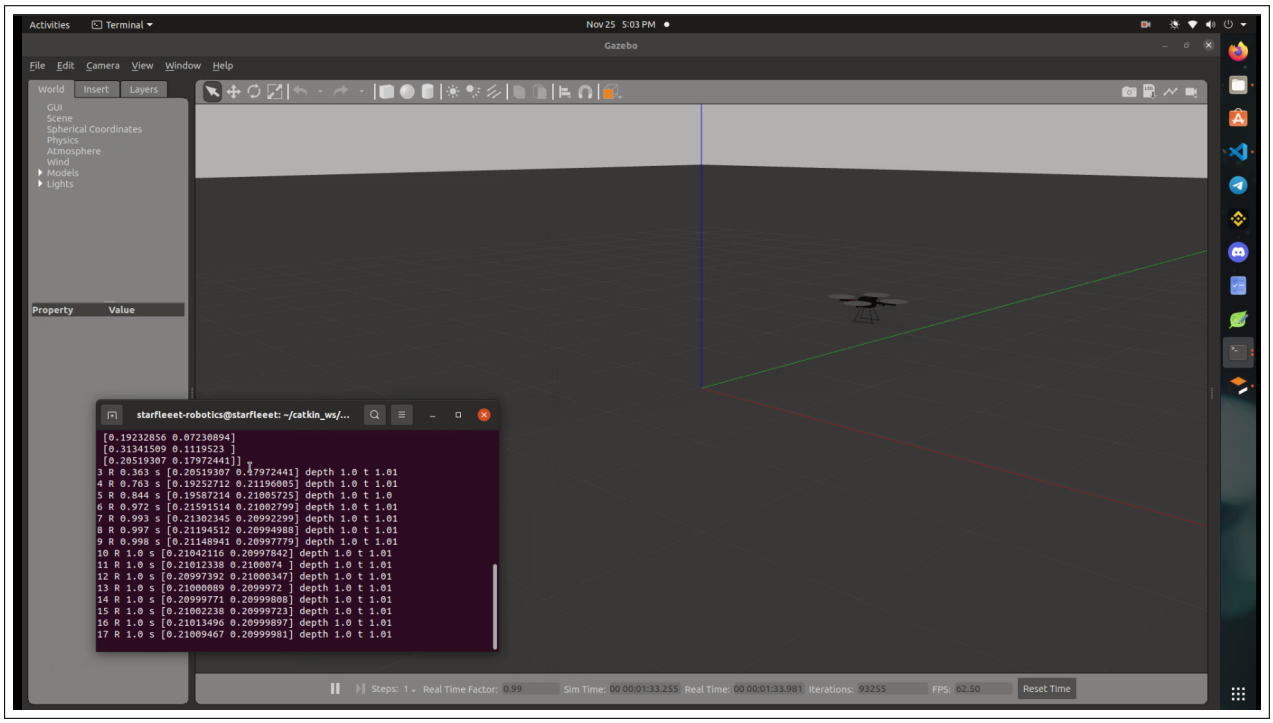


Figure 5.8: Gazebo simulation of Double Q-PID controlled Hector Quadroto Robot

5.3 Simulation instructions to run the Robots with D-QPID Algorithm

Please refer our github repository for instructions to simulate the algorithm and to verify the results.
Github Link : Double-QPID Algorithm

```
import numpy as np
from collections import deque
import signal
import time
import pickle
# custom files
from Algorithm.action_chooser import action_chooser
from Algorithm.double_Q import DQPID
from Algorithm.memory import memory_comparator
from Algorithm.functions import sigint_handler
from Algorithm.algorithm_4 import algorithm_4
from Algorithm.algorithm_3 import algorithm_3
from Algorithm.algorithm_2 import algorithm_2
from robot_dict import robot_dict as robot_dict
from Algorithm.long_term_memory import update_long_term_memory

# ROBOT
# select the platform from the robot dictionary
# For husky ---> husky_pi_random
# For Hector drone ---> drone_pi
platform = 'drone_pi'
# the pioneer and other robots have different ros topic namess
# with this you can select the correct one
simulation = True
# this sets the mode for the memory
mode = 'long_term_memory'
```

```
# Constants
E_GREED = 1.
EXECUTION_TIME = 400
Ts = 10.

Teval = 1.
N_memories = 8
memory_repetition = 8
np.random.seed(1234)

# tracking and velocity control
def main_DQPID(load):
    global platform
    signal.signal(signal.SIGINT, sigint_handler) # to execute the signal ...
       interrupt


    # QPID parameters
    Q_index = 0
    Q_arrange = deque()
    action_discretization_n = 3
    maximum_depth = 8

    # robot parameters
    current_robot = robot_dict[platform]
    state = current_robot['initial_state']
    set_point = current_robot['set_point']
    initial_action_centroid = current_robot['action_centroid']
    robot = current_robot['class'](set_point, dt = 1./Ts, Teval=Teval, ...
       simulation=simulation)
    K_step = current_robot['K_step']
    print('k step', K_step)
    time.sleep(1)
    # Classes instantiations
    action_selector = action_chooser(E_GREED, EXECUTION_TIME)
    memory = memory_comparator(memory_repetition)
    Q_arrange.append(DQPID(state, None,1.,initial_action_centroid, ...
       action_discretization_n, maximum_depth, 0., 0., K_step=K_step  ))
    ltm = deque(maxlen = 10000) # long term memory max
    minibatch_size = 32
    # others
    time.sleep(1)
    state = state[0]
    if load:
        n_to_load = np.load('len_Q.npy')[0]
        for _ in range(n_to_load):
            file = open('Q_arrange' + str(_) + '.txt','r')
            Q_arrange[_] = pickle.load(file)

    # I propose the generation of a new class, that will be responsible ...
       for saving everything and plotting

    for x in range(EXECUTION_TIME):

        start = time.time()

        flag_ab, action, action_index, e_greed, state_index = ...
           action_selector.get(Q_arrange[Q_index], state)
        memory.update(state_index, action_index, Q_index, action)

        next_state = robot.update(action, Q_arrange[Q_index].depth)
```

```
        reward = robot.get_gaussian_reward(next_state,set_point)

        if x < N_memories:
             Q_arrange[Q_index] = algorithm_2(Q_arrange[Q_index],state_index,
             next_state,reward,action_index, flag_ab)
             next_Q_index = Q_index
             memory.counter = 0

        else:
            memory.compare()
            if memory.flag_no_variation == True:
                #print('algorithm 4')
                Q_arrangement, Q_index, next_Q_index, ...
                    action_selector.e_greed_counter = ...
                    algorithm_4(Q_arrange,memory.Mt,next_state,
                reward,maximum_depth,action_discretization_n,
                action_selector.e_greed_counter,set_point, flag_ab, K_step)
                memory.flag_no_variation = False
            else:
                #print('algorithm 3')
                Q_arrange, Q_index, next_Q_index = ...
                    algorithm_3(Q_arrange,memory.Mt,next_state,
                reward,flag_ab)

        end = time.time()

        ltm.append([state, next_state, action, Q_index, next_Q_index, ...
            reward, flag_ab, action_index])
        if mode=='long_term_memory':
            Q_arrange = update_long_term_memory(ltm, Q_arrange, ...
                minibatch_size)

        Q_index = next_Q_index
        state = next_state

        print(x, 'R',  round(reward,3),'s', next_state, 'depth', ...
            Q_arrange[Q_index].depth, 't', round(end-start,2) )



    #saving Q tables
    for _ in range(len(Q_arrange)):
        file = open('Q_arrange' + '.txt','wb')
        pickle.dump(Q_arrange[_], file)

    np.save('len_Q.npy', np.array([len(Q_arrange)]))
    robot.plotter.save_values()
    mse = robot.plotter.mean_squared_error(set_point)
    print('mse', mse, 'mean mse', np.mean(mse))
    euclidean_distance = robot.plotter.euclidean_distance(set_point)
    print('euclidean_distance', euclidean_distance)
    mahalanobis = robot.plotter.mahalanobis(set_point)
    print('mahalanobis', mahalanobis)

    # ploting and printing performance
    print('actions', action)
    robot.plotter.plot(savefig=True)
    robot.plotter.save_values()
    mse = robot.plotter.mean_squared_error(set_point)
    print('mse', mse, 'mean mse', np.mean(mse))
```

```
    euclidean_distance = robot.plotter.euclidean_distance(set_point)
    print('euclidean_distance', euclidean_distance)
    mahalanobis = robot.plotter.mahalanobis(set_point)
    print('mahalanobis', mahalanobis)


    robot.stop()



if __name__ == '__main__':
    main_DQPID(load = False)
```

5.4 Future work

The future scope of this project might be implementing a better mechanism for the replay buffer. Instead of using stochastic experience replay, we can use combined experience replay (CER)[3] will help to sample memories efficiently and increase the overall performance. Using a deep neural network will increase the overall optimization process by updating the weights of PID gains using deep reinforcement learning techniques have become the current state of the art for any task revolving around optimization.

References

[1] Kiam Heong Ang, Gregory Chong, and Yun Li. "PID control system analysis, design, and technology." In: *IEEE transactions on control systems technology* 13.4 (2005), pp. 559–576.

[2] Ignacio Carlucho et al. "Incremental Q-learning strategy for adaptive PID control of mobile robots." In: *Expert Syst. Appl.* 80 (2017), pp. 183–199.

[3] Hongyang Dong and Xiaowei Zhao. "Composite Experience Replay-Based Deep Reinforcement Learning With Application in Wind Farm Control." In: *IEEE Transactions on Control Systems Technology* (2021).

[4] Hado Hasselt. "Double Q-learning." In: *Advances in neural information processing systems* 23 (2010), pp. 2613–2621.

[5] Rodrigo Hernández-Alvarado et al. "Neural network-based self-tuning PID control for underwater vehicles." In: *Sensors* 16.9 (2016), p. 1429.

[6] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG].

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

[8] Neha Tandan and Kuldeep Kumar Swarnkar. "PID Controller Optimization By Soft Computing Techniques-A." In: *International Journal of Hybrid Information Technology* 8.7 (2015), pp. 357–362.

[9] Qing-Guo Wang, Ho-Wang Fung, and Yu Zhang. "PID tuning with exact gain and phase margins." In: *ISA Transactions* 38.3 (1999), pp. 243–249. ISSN: 0019-0578. DOI: https://doi.org/10.1016/S0019-0578(99)00020-8. URL: https://www.sciencedirect.com/science/article/pii/S0019057899000208.

[10] Shangtong Zhang and Richard S. Sutton. *A Deeper Look at Experience Replay*. 2018. arXiv: 1712.01275 [cs.LG].