5-2024

Deep Reinforcement Learning of Variable Impedance Control for Object-Picking Tasks

Akshit Lunia
alunia@g.clemson.edu

Deep Reinforcement Learning of Variable Impedance Control for Object-Picking Tasks

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Mechanical Engineering

by
Akshit Lunia
May 2024

Accepted by:
Dr. Yue Wang, Committee Chair
Dr. John Wagner
Dr. Phanindra Tallapragada

Abstract

The increasing deployment of robots in industries with varying tasks has accelerated the development of various control frameworks, enabling robots to replace humans in repetitive, exhaustive, and hazardous jobs. One critical aspect is the robots' interaction with their environment, particularly in unknown object-picking tasks, which involve intricate object weight estimations and calculations when lifting objects. In this study, a unique control framework is proposed to modulate the force exerted by a manipulator for lifting an unknown object, eliminating the need for feedback from a force/torque sensor. The framework utilizes a variable impedance controller to generate the required force, and an admittance controller models the robot's motion as a mass-spring-damper system. The combined framework mimics a human hand guiding a robot arm, where the force generated by the variable impedance controller pulls the robot to the desired position. The distance to the desired position, stiffness, and damping parameters influence the variable impedance force generated. The stiffness and damping parameters are uniquely tailored for specific object masses and require learning. Here, deep reinforcement learning is employed to learn the stiffness parameter, enabling the framework to lift objects of unknown mass effectively. The effectiveness of the proposed control framework is demonstrated through training and testing in the ROS Gazebo simulator, employing a UR5 manipulator. The trained model exhibits the ability to lift objects with unknown masses to predetermined positions, showcasing the framework's practical applicability and potential in diverse industrial settings.

Dedication

This thesis is a tribute to those who shaped my path. I dedicate this to my family, partner, and friends, your unwavering support fuels my journey. To my adviser Dr. Yue Wang for the opportunities and constant support. Thank you.

Acknowledgments

I would like to express my deepest gratitude to the exceptional individuals who supported and guided me throughout this transformative journey. I am deeply thankful for the guidance, support, and valuable insights provided by my adviser, Dr. Yue Wang. Your expertise and encouragement have been instrumental in shaping the direction of this research. I highly valued the weekly meetings we held, which not only served as crucial checkpoints to keep me on track academically but also provided me with plenty of encouragement.

I also appreciate the resources and facilities provided by the Interdisciplinary Intelligence Research (I^2R) Lab and Clemson University, which have been essential for conducting the experiments and gathering the necessary data for this thesis. I would especially like to thank the support offered by my colleague at I^2R lab, Mr. Zhanrui Liao.

My heartfelt thanks go to my family and friends for their unwavering support, understanding, and encouragement throughout this challenging yet rewarding journey. Your belief in me has been a constant source of motivation.

In conclusion, completing this thesis would not have been possible without the support and encouragement of these wonderful individuals and institutions. Thank you for being an integral part of this academic journey

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Manipulator Object-Picking Task

Ever since the beginning of robotics, researchers have been experimenting with ways to imitate human behaviors with robots. One of the main behaviors of focus is being able to manipulate objects. Object manipulation is one of the basic human activities, and with robots being introduced in different industries like manufacturing, medicine, warehouses, and more, being able to interact with their environment is imperative. When observing an object-picking task, commonly known as an object pick and place task, humans perform a multitude of estimations and calculations. These include object weight, trajectory and path planning, and grasping mechanics. Being able to interact with objects and manipulate them the way humans do will enable robots to be readily introduced to human workplaces and replace them in repetitive, harmful, and exhausting applications.

Robots are skilled at grasping and manipulating objects in repetitive, familiar settings such as industrial setups. The objects' material properties, geometry, and weight are controlled and known in such settings. The robots can handle some variations in object properties, but the whole process is typically optimized to a limited set of expected variations [?]. Early factory settings employed robot arms to follow predetermined trajectories, assuming the objects would appear at the exact predefined location. With the advancement in machine learning and control algorithms, the robots can now adapt to changes in object location and generate appropriate trajectories governed by the laws set by the control algorithms, allowing humans to drop the object in the vicinity of the robot or on a conveyor without being specific on the location of it. The current industries require solutions that can be deployed for varying objects where the objects' rigidness, shape, weight, and other properties are not known entirely. A control algorithm that can adapt to such variances in object properties is desired here. Two main problems must be solved when working with objects: grasping and manipulation.

The grasping problem contains complexities like object detection, object properties, grasping position and force. Detecting objects is a challenge in robotics that demands high precision across a wide range of objects, even for basic tasks like object-picking. Researchers have devised unique algorithms, drawing from various sources and sensor technologies, to tackle this issue. In [40], A. Okamura and M. Cutkosky proposed a method to enhance detection accuracy by incorporating multiple viewing angles and high-resolution images. Extensive datasets were explored to train classifiers and the probabilistic fusion of outputs from multiple object detectors to boost accuracy. Additionally, pan-tilt-zoom cameras were introduced to capture detailed views of objects. The authors demonstrated that their probabilistic approach significantly enhanced accuracy when detecting objects from various perspectives. The effectiveness of training classifiers was also showcased on large synthetic datasets, resulting in high-performance object detection.

Furthermore, in [13], A. Coates and A. Ng addressed the challenge of combining classifiers for different viewpoints, highlighting the complexities of detecting object classes from diverse angles reliably. The work suggests employing multiple cameras and high-resolution imagery to validate and enhance object detection accuracy. Object detection algorithms typically use neural networks to identify an object. A learning pipeline was then introduced to integrate offline and online learning to swiftly train robots to detect new objects within a few seconds. The challenges were tackled by applying deep learning models to robotics, particularly in localizing the bounding box around an object and assigning its label. The suggested pipeline capitalized on merging a feature extraction module trained offline with a region classifier trained online, enabling rapid adaptation to new objects. The readily available object detection algorithms identified the object class well and were robust enough for real-world applications.

The sense of touch provides diverse sensory information, including vibration, pressure, and temperature, aiding humans in perceiving their environment [36, 42, 44]. While research on object property detection is well-documented, it often requires additional sensors, typically tactile sensors, to identify physical attributes. In their work [37], E. Maiettini et al. investigate an approach for haptic exploration of unknown object surfaces using robotic fingers. They define features based on local surface curvature and introduce algorithms for feature detection using a spherical fingertip equipped with a tactile sensor. The haptic exploration aims to discern object shape, texture, and other physical attributes. Once the object and its properties are identified, the subsequent step involves determining the grasping position and force.

In [16], N. Doshi et al. discuss a novel approach to manipulating unknown objects by regulating the object's contact configuration with the robot and the environment. They estimate the robot's wrench and motion constraints to manipulate different objects. Similar works on the grasping problem are being carried out in [43] and [53]. Authors in [43] develop a vision-based grasping system that uses range data to find grasp points for objects of varying shapes. In [53], a methodology is introduced to calculate the grasping force necessary to lift and manipulate objects with minimum deformation. They use deformation and slipping data to estimate the grasping force. These techniques are crucial in successfully grasping and manipulating rigid and soft objects. The techniques described here focus on grasping and manipulation by estimating the object's mass and material properties with the help of various sensors. The research regarding manipulating objects of unknown mass is limited to tackling the grasping problem, focusing on the force required to grasp the object with various state-of-the-art sensors and rarely discussing the effort required by the robot arm to manipulate the object of unknown mass.

Robots with additional sensors for appropriate environment and object detection are expensive and require frequent calibrations, resulting in an undesirable increase in the working cost and the initial investment. This thesis proposes a control framework trained to reach and lift an object of unknown mass without using a force/torque sensor, typically used in other techniques to estimate object mass. The proposed framework mimics the human behavior of adjusting the force applied to lift an object of unknown mass based on its initial observations. We deploy three main concepts to achieve this: a variable impedance controller, an admittance controller, and a deep reinforcement learning (DRL) algorithm. Variable impedance controller learned using DRL is responsible for generating the force lifting the object of unknown mass, whereas admittance controller converts the lifting force into acceleration. First, we will introduce the general concepts of impedance control, admittance control, and deep reinforcement learning to develop a background in Sections 1.2, 1.3, and 1.4, respectively. Then, we discuss the object-picking problem in Chapter 2 and delve further into the three main components of our control framework concerning the object-picking task. In Chapter 3, we derive our manipulator control laws and convert the object-picking problem into a DRL problem. Further, in Chapter 4, we simulate our framework to train the agent in learning appropriate policy. Finally, in Chapter 5, we discuss our observed results and compare the trained proposed policy with a fixed impedance controller and a variable PD controller trained using the same DRL algorithm.

1.2 Impedance Control

Robotic manipulators have been successfully applied in simple manipulation applications such as sliding [48], throwing [20], pivoting [6], spray painting and arc welding, where the manipulator must only follow a position trajectory [23]. The difficulty arises when robots are required to perform contact-rich actions, such as polishing and assembly tasks, and/or operate in unknown environments. Robots needed in real-world applications such as in industries, healthcare, and households [5] must be able to control the interaction forces and motion carefully. Both motion and force controllers for robotic manipulators have been widely researched and developed [51] [63]. Though there are several approaches, we can classify them into two significant categories [12]: impedance control [24] and hybrid position/force control [41].

Hybrid position/force controller controls simultaneously and independently force and position parameters [1]. It generates force in one axis while motion in the others, or vice versa [25]. The general hybrid position/force controller can be seen in Figure 1.1.

general_hybrid_controller.png

Figure 1.1: General Hybrid Position/Force Control Structure [1].

The vectors v and f respectively represent the robot's velocity and force exerted by it in either cartesian or joint coordinates. Vectors v_{des} and f_{des} are the desired respective velocity and force vectors. Hybrid position/force controllers are deployed in applications where the force and motion can be separated between the axes. For example, let us take a manipulator robot trying to clean a whiteboard with an eraser. The manipulator applies force against the board to maintain appropriate contact force while having motions along the plane of the whiteboard (Refer to Figure 1.2). This shows how the force and motion are separated between the axes when using a hybrid position/force controller. The effectiveness of the hybrid position/force controllers can also be found in detail for various other such applications [58, 60, 62, 14, 61].

robot_erasing.png

Figure 1.2: Manipulator applies force in the z axis and has motion in the x and y axes while erasing a whiteboard.

On the other hand, impedance control provides a unified control law that combines force and motion and does not separate them into different axes. Impedance control models the interaction force as a mass-spring-damper system, whereby depending on the perceived force between the robot and its environment, the robot modifies its motion to either increase or decrease the interaction force [24]. Impedance control is an indirect force controller that seeks to control the impedance property instead of the actual position or force in the manipulator-object interface during interaction [57].

The idea behind designing the impedance control as a mass-spring-damper system is to imitate human musculoskeletal structure, where we change the stiffness of our muscles to vary the forces we apply to our environment. Observe Figure 1.3; the robot is tasked to reach the desired position ($\boldsymbol{x}_o$), which the impedance control will convert the desired motion into force and moves while interacting with the plant dynamics. The interaction force ($\boldsymbol{F}_{ext}$) is measured and used as feedback by the impedance controller.

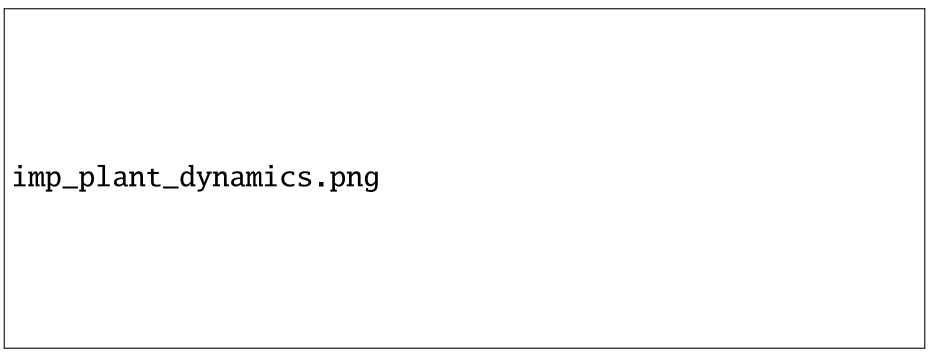

Figure 1.3: Implementation of Impedance Control.

There are two types of impedance control when considering a manipulator object pick-up task: object impedance control and robot impedance control. Robot impedance control models the robot dynamics as a compliant system wherein the robot mimics a mass-spring-damper system. In the case of object impedance control, the object held by the robot is modeled to mimic the mass-spring-damper system [47]. The motion and force interaction of the object with its environment is essential here. Some applications of object impedance control can be found in collaborative manipulation of an object between humans and robots, such as in [46]. Though we will be using impedance control to manipulate an object, we are not interested in the object's interaction forces with its environment. Instead, we use an impedance controller to generate a force that pulls on the object. We will further explore this idea in Section 2.1.

Impedance control in most applications is used in cartesian space to control the end-effector interaction with the environment [34, 4, 49, 11], as observed in haptic exploration [17], but can also be derived to be used in joint space [55]. Impedance control is crucial when robots interact with stiff environments and for new robot applications that bring humans and robots to share spaces, making contact between them inevitable [3]. Hence, it becomes essential to ensure human safety [21], making impedance control an indispensable tool. When working alongside humans, the robots are not only supposed to be in the human's space and perform some specific tasks but also assist humans in various tasks such as co-manipulation of heavy object [46, 26], handover objects [9], and various other collaborative tasks. When robots are deployed in environments where they need to interact with multiple entities or perform different tasks within their environment, such as opening/-closing a door, turning on/off switches, carrying objects, etc., it becomes necessary for the robot to be able to modulate its impedance to be able to apply appropriate force to complete the task. This modulation in impedance is popularly known as variable impedance control, wherein the impedance parameters such as mass, stiffness, and damping parameters can be varied to achieve desired compliance. Variable impedance control is widely preferred in such tasks [2, 45].

As discussed, impedance control is quite effective in modeling the force interaction between the robot and its environment. In our application, the interactive force is interpreted as a phantom force required to lift an object of unknown mass. This interpretation allows us to modulate the force based on the observed varying object displacement when the robot applies the lifting phantom force.

1.3 Admittance Control

Similar to an impedance controller, admittance control models the force as a mass-spring-damper system but uses the force applied by the environment as an input and generates motion corresponding to the applied force (Refer to Figure 1.4). The design of the admittance controller impacts the robot's reaction to the applied force. We can make the robot highly reactive by decreasing the damping and stiffness. Similarly, we can reduce the reactiveness by increasing the stiffness and damping, allowing us to achieve our desired response behavior [33, 38].

add_plant_dynamics.png

Figure 1.4: Implementation of Admittance Control.

This type of control is widely used in collaborative manipulation tasks [46] and haptic interaction [18], wherein the human can pull on the object held by the robot and human, and the force is transmitted via the object to the robot. Then, the admittance controller generates motion in the robot along the force. Admittance control was first introduced on retrofitted robots exploiting the force sensor at the base of the robot to increase safety when working in an industrial capacity [31]. In [22], S. Grafakos et al. develop a control framework that uses electromyography data of the human muscle arm to vary the damping in the admittance controller, enabling higher cooperative movement accuracy and reduction in human effort. In [54], S. Tarbouriech et al. propose a control strategy for collaborative manipulation between humans and dual-arm robots. They deploy an admittance control to move the object within the workspace, and they also use gravity compensation to cancel the object's gravity effects. C. Yang et al. in [59] develop an admittance control method that adapts to the unknown dynamics of its environment using an adaptive neural network, ensuring the robot achieves the desired trajectory. Often, when using admittance control in human-robot cooperative tasks, it is essential to estimate the human's intent to model appropriate admittance control response. In [29], G. Kang et al. develop different admittance controller responses along direct or indirect human intention. The direct human intention admittance controller provides a rapid response to human force, whereas the indirect human intention admittance controller is used to minimize the trajectory error in long-term tasks.

The applications of admittance control are vast, especially in human-robot co-manipulation. Admittance controllers are also used to model the interaction between the environment and robot end-effector in cases where the robotic system does not provide access to low-level control, such as control over joint torque [56]. In our application, we face a similar issue where the manipulator does not provide access to control over joint torque. Hence, the phantom force generated by the impedance control must be converted into velocity/position inputs for the manipulator using admittance control.

1.4 Deep Reinforcement Learning

Humans are versatile in adapting to highly unpredictable and uncertain scenarios. In comparison, classical robotics requires a highly constrained environment to perform a particular task using high-gain negative error feedback controllers. Robots need a compliant low-gain control capable of estimating appropriate actions for a dynamic task to adapt to different scenarios and uncertainties.

Reinforcement learning (RL) is a widely used solution in robotics to overcome such dynamic environments. RL is essentially learning through interaction [7]. An RL agent interacts with its environment and observes the consequences of its actions [7, 28]. According to the observed consequences, the agent learns and alters its behavior to achieve the maximum reward provided by a reward function. A reward function is a mathematical equation defining a task's success or failure when performing a specific action. Using the reward function, the agent explores the environment by performing actions (a_t) and observing the change in the state (s_t) of the environment. The reward function then uses the observations to provide a reward (r_t). The idea is similar to training a pet; we provide positive reinforcement as a treat when the pet performs an action that we want it to do. An RL agent, after performing multiple actions and generating rewards for those actions, starts learning a policy (π) that will enable it to find an optimal solution to maximize the reward it receives (Refer to Figure 1.5).

rl.png

Figure 1.5: Reinforcement Learning Workflow.

Essentially, the Markov decision process (MDP) is used to describe RL [7] consisting of a set of states ($\boldsymbol{S}$), a set of actions ($\boldsymbol{A}$), a transition dynamics ($\boldsymbol{T}(s_{t+1}|s_t, a_t)$) that map a state-action pair at time t onto a distribution of states at time $t+1$, an immediate reward function ($\boldsymbol{R}(s_t, a_t, s_{t+1})$), and a discount factor ($\gamma \in [0, 1]$). The lower values of the discount factor (γ) provide more weight to the immediate rewards. The policy (π) maps the states to a probability distribution over action,

$$\pi : \boldsymbol{S} \rightarrow p(\boldsymbol{A} = a|\boldsymbol{S}) \tag{1.1}$$

The goal of RL is to find an optimal policy (π^*) that provides the maximum reward from all states,

$$\pi^* = \operatorname{argmax}_\pi \mathbb{E}[\boldsymbol{R}|\pi] \tag{1.2}$$

There are three approaches to solving RL problems: methods based on value functions, methods based on policy search, and methods that employ both value functions and policy search, commonly known as the hybrid actor-critic approach. Value function methods require estimating the value of being in a particular state. Policy search methods do not need a value function and instead directly search for an optimal policy π^*. Actor-critic methods combine value function and policy search methods, as shown in Figure 1.6. The "actor" (policy) learns by using the feedback from the "critic" (value function). The actor-critic method aims to solve the problems faced by value function and policy search methods, trading off variance reduction of policy gradients with bias introduction from value functions methods [32, 50].

Figure 1.6: The actor-critic setup.

In [35], J. Luo et al. use RL to learn the variable impedance controller for a tight-fit assembly. The assembly consisted of four sequential steps requiring high accuracy, which is beyond a typical industrial robot. Using RL with variable impedance control, they achieved the skills to assemble by mapping the interaction forces to control actions. J. Buchli et al. have a similar approach in [10], creating a framework that scales to complex robotic systems while learning both the appropriate trajectory and the time-varying impedance control. RL tasks can be significantly simplified by carefully designing the action and observation spaces. This concept of simplifying RL is explored by R. Martín-Martín et al. in [39], wherein they showcase the result of RL training by selecting a simplified action space.

Although RL has succeeded in various applications and fields, it lacks scalability and is inherently limited to low-dimensional problems [7]. These limitations exist in RL algorithms, similar to other algorithms, and contain complexity issues such as memory complexity, computational complexity, and sample complexity in the case of machine-learning algorithms [52]. Deep learning can be helpful here with its ability to automatically find low-dimensional representations of high-dimensional data [7]. Deep learning enables RL to scale decision-making problems and simplify policy learning for model-free applications by reducing memory, computational, and sample complexities. Deep learning with RL is often dubbed deep reinforcement learning (DRL).

DRL combines an artificial neural network with reinforcement learning to map the actions to states and generate a policy function. The main difference between RL and DRL is using artificial neural networks to approximate the optimal policy (π^*) and/or the optimal value functions [7]. In RL, we create a table of values for each action performed at a particular state. This data table can be enormous in continuous environments, which is usually true in robotics and the real world. Instead, DRL uses artificial neural networks that learn to map actions to states and estimate the value of a particular action for a specific state. Using DRL, we can create a control framework that can adapt to and learn a dynamic environment and task.

Our application uses DRL to learn the optimal policy necessary to generate the phantom force (as introduced in Section 1.2). The optimal policy should be able to observe the current state of the robot arm and the desired goal and generate the necessary impedance parameters to move the robot arm from its current position to the desired goal while holding the object of unknown mass. In Chapter 2, we dive deeper into the object-picking task, variable impedance control, admittance control, and twin-delayed deep deterministic policy gradient.

Chapter 2

Problem Statement

Consider an object-picking task where the object mass ($\boldsymbol{m}$) is unknown and varies with each successful task completion. The end-effector and object locations vary in every task episode along with the object mass. The objective of the task is for the end-effector arm to reach the object location, grasp it, and apply the appropriate force necessary to lift the object of unknown mass to the desired goal location without using an F/T sensor or any object mass measurement.

When tasked with lifting an object of unknown mass to a certain position, we first estimate its mass based on our previous experience of lifting it. If our estimation is inaccurate, we modulate the force we apply to lift and move the object toward the goal. The modulation of force is a necessary ability when lifting an object with an unknown mass. For robots, impedance control is a popular control technique used to generate the force that the manipulator applies on its environment during interaction, in this scenario, the object. Impedance control force is a function of distance to the goal and will modulate force generated based on the end-effector's distance to the desired goal and not the object mass. So for varying object mass, we require varying impedance control wherein by varying the stiffness ($\boldsymbol{K}_d(t) \in 6 \times 6$) and damping ($\boldsymbol{D}_d(t) \in 6 \times 6$) matrices, we can generate the force

($\boldsymbol{W}^c \in 6 \times 1$) required to lift an object with different masses.

The UR5 manipulator arm is either a velocity-controlled or a position-controlled robot and does not accept force as an input. This is a common problem in robotics, and we solve this using an Admittance Controller, which converts the force acting on the robot into robot motion. Here, the variable impedance force acts like a phantom force that pulls the robot towards the desired goal position. The Admittance Controller converts the phantom force ($\boldsymbol{W}^c$) into end-effector acceleration ($\ddot{\boldsymbol{x}}^A \in 6 \times 1$). The end-effector acceleration ($\ddot{\boldsymbol{x}}^A$) is then converted into the end-effector position ($\boldsymbol{x}^c$ using kinematic equations.

2.1 Variable Impedance Control for Object-Picking Task

Impedance control is a control technique that provides a relationship between position, velocity, acceleration, and force, all four, instead of controlling just one of the state variables [8]. Impedance control allows us to model the robot as a mass-spring-damper system. And like a mass-spring-damper system, we can make the robot compliant or stiff. Let's take a manipulator arm that needs to reach a certain desired end-effector position (refer to Figure 2.1). When moving toward its desired position, the manipulator arm will apply a certain force to its environment when opposed, called $\boldsymbol{F}_{ext}$. To avoid this force from damaging the robot or its environment, we model the interaction force as a mass-spring-damper system, which reduces the overall force applied by the robot arm when trying to reach the goal. The mass-spring-damper system is a function of its stiffness and damping parameters, and by changing them, we can change the system's behavior. The same principle can be applied to an impedance controller where by varying the stiffness ($\boldsymbol{K}_d(t)$) and damping ($\boldsymbol{D}_d(t)$) parameters we can create a variable impedance controller.

imp.png

Figure 2.1: Impedance External Force Illustration.

In this section, we derive the task space variable impedance control [27]. The equation of motion of the robot is,

$$\boldsymbol{\tau} = \boldsymbol{M(q)}\ddot{\boldsymbol{q}}^{m} + \boldsymbol{C(q, \dot{q})}\dot{\boldsymbol{q}}^{m} + \boldsymbol{g(q)} + \boldsymbol{J}^{T}(\boldsymbol{q})\boldsymbol{F}_{ext} \tag{2.1}$$

Where, $\boldsymbol{q}$ is the joint angular position (6×1), $\dot{\boldsymbol{q}}$ is the joint angular velocity (6×1), $\ddot{\boldsymbol{q}}$ is the joint angular acceleration (6×1), $\boldsymbol{\tau}$ is the joint actuation torque, $\boldsymbol{M(q)}$ is the inertia matrix (6×6), $\boldsymbol{C(q, \dot{q})}$ is the Coriolis matrix (6×6), $\boldsymbol{g(q)}$ is the gravity matrix (6×1), and $\boldsymbol{J}^{T}(\boldsymbol{q})\boldsymbol{F}_{ext}$ is the external torque wrenches.Here $\boldsymbol{M(q)}$, $\boldsymbol{C(q, \dot{q})}$, and $\boldsymbol{g(q)}$ can be calculated using equations (2.2, 2.4, 2.3) [30].

$$\boldsymbol{M(q)} = [\sum_{i=1}^{n}(m_i J_{v_i}^{T}\boldsymbol{J}_{v_i} + \boldsymbol{J}_{w_i}^{T} R_i I_i R_i^{T} \boldsymbol{J}_{w_i})] \tag{2.2}$$

where, $\boldsymbol{J}_{v_i}$ and $\boldsymbol{J}_{w_i}$ are the respective linear and angular parts of the Jacobian matrix $\boldsymbol{J}_i$. For the coriolis matrix, we derive its elements (c_{ij}) from the elements of the inertia matrix (m_{ij}) via the formula,

$$c_{ij} = \sum_{k=1}^{n} \frac{1}{2}(\frac{\partial m_{ij}}{\partial q_k} + \frac{\partial m_{ik}}{\partial q_j} + \frac{\partial m_{kj}}{\partial q_i})\dot{q}_k \tag{2.3}$$

Finally, the elements of the gravity vector ($g_i(q)$) are given by,

$$g_i(q) = \frac{\partial \mathcal{P}}{\partial q_i} \tag{2.4}$$

Here, $\mathcal{P}$ is the potential energy due to gravity. Since impedance controller models external interaction force as a mass-spring-damper system,

$$\boldsymbol{J}^T(\boldsymbol{q})\boldsymbol{F}_{ext} = \boldsymbol{K}_d(\boldsymbol{q})(\boldsymbol{q}_d - \boldsymbol{q}^m) + \boldsymbol{D}_d(\boldsymbol{q})(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}^m) + \boldsymbol{M}_d(q)(\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}^m) \tag{2.5}$$

Here, $\boldsymbol{q}_d$ is the desired joint angular position (6×1), $\dot{\boldsymbol{q}}_d$ is the desired joint angular velocity (6×1), $\ddot{\boldsymbol{q}}_d$ is the desired joint angular acceleration (6×1), $\boldsymbol{K}_d(\boldsymbol{q})$ is the desired variable joint space stiffness matrix (6×6), $\boldsymbol{D}_d(\boldsymbol{q})$ is the desired variable joint space damping matrix (6×6), and $\boldsymbol{M}_d(\boldsymbol{q})$ is the desired joint space inertia matrix. By substituting Equation (2.5) in (2.1) we get,

$$\boldsymbol{\tau} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}}^m + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}}^m + \boldsymbol{g}(\boldsymbol{q}) + \boldsymbol{K}_d(\boldsymbol{q})(\boldsymbol{q}_d - \boldsymbol{q}^m) + \boldsymbol{D}_d((\boldsymbol{q}))(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}^m) + \boldsymbol{M}_d(\boldsymbol{q})(\ddot{\boldsymbol{q}}_d - \ddot{\boldsymbol{q}}^m) \tag{2.6}$$

We can set the desired inertia matrix as the actual inertia matrix to simplify the equation of motion. Therefore,

$$\boldsymbol{\tau} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}}_d + \boldsymbol{C}(\boldsymbol{q},\dot{\boldsymbol{q}})\dot{\boldsymbol{q}}^m + \boldsymbol{g}(\boldsymbol{q}) + \boldsymbol{K}_d(\boldsymbol{q})(\boldsymbol{q}_d - \boldsymbol{q}^m) + \boldsymbol{D}_d((\boldsymbol{q}))(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}^m) \quad (2.7)$$

Since we are interested in the interaction between the end-effector and the object as well as the distance of the end-effector to the goal location, we formulate the problem in the task space instead of the joint space. According to differential kinematics, we know

$$\dot{\boldsymbol{q}} = \boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}} \quad (2.8)$$

Where $\dot{\boldsymbol{x}}$ is the end-effector velocity (6×1), and $\boldsymbol{J}(\boldsymbol{q})$ is the Jacobian matrix (6×6). On differentiating Equation (2.8), we get

$$\ddot{\boldsymbol{q}} = \boldsymbol{J}^{-1}(\boldsymbol{q})\ddot{\boldsymbol{x}} - \boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{J}}(\boldsymbol{q})\boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}} \quad (2.9)$$

Also, joint actuation torque can be converted to task-space force as,

$$\boldsymbol{W}^c = \boldsymbol{J}^T(\boldsymbol{q})\boldsymbol{\tau} \quad (2.10)$$

On substituting Equations (2.8), (2.9), and (2.7) in Equation (2.10), we get task space equation of motion as,

$$\begin{aligned} \boldsymbol{W}^c = {} & \boldsymbol{K}_d(t)(\boldsymbol{x}_d - \boldsymbol{x}^m) + \boldsymbol{D}_d(t)(\dot{\boldsymbol{x}}_d - \dot{\boldsymbol{x}}^m) + \boldsymbol{J}^{-T}(\boldsymbol{q})\boldsymbol{M}(\boldsymbol{q})\boldsymbol{J}^{-1}(\boldsymbol{q})\ddot{\boldsymbol{x}}_d \\ & + \boldsymbol{J}^{-T}(\boldsymbol{q})[\boldsymbol{C}(\dot{\boldsymbol{q}},\boldsymbol{q}) - \boldsymbol{M}(\boldsymbol{q})\boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{J}}(\boldsymbol{q})]\boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}}^m \\ & + \boldsymbol{J}^{-T}(\boldsymbol{q})\boldsymbol{g}(\boldsymbol{q}) \end{aligned} \quad (2.11)$$

Let,

$$\boldsymbol{\Lambda}(\boldsymbol{x}) = \boldsymbol{J}^{-T}(\boldsymbol{q})\boldsymbol{M}(\boldsymbol{q})\boldsymbol{J}^{-1}(\boldsymbol{q})$$

$$\boldsymbol{\mu}(\dot{\boldsymbol{x}}, \boldsymbol{x}) = \boldsymbol{J}^{-T}(\boldsymbol{q})[\boldsymbol{C}(\dot{\boldsymbol{q}}, \boldsymbol{q}) - \boldsymbol{M}(\boldsymbol{q})\boldsymbol{J}^{-1}(\boldsymbol{q})\dot{\boldsymbol{J}}(\boldsymbol{q})]\boldsymbol{J}^{-1}(\boldsymbol{q})$$

$$\boldsymbol{\gamma}(\boldsymbol{x}) = \boldsymbol{J}^{-T}(\boldsymbol{q})\boldsymbol{g}(\boldsymbol{q})$$

where, $\boldsymbol{\Lambda}(\boldsymbol{x})$ is the task space Inertia matrix (6×6), $\boldsymbol{\mu}(\dot{\boldsymbol{x}}, \boldsymbol{x})$ is the task space Coriolis matrix (6×6), and $\boldsymbol{\gamma}(\boldsymbol{x})$ is the task space gravity matrix (6×1). Therefore, the task space variable impedance control is,

$$\boldsymbol{W}^c = \boldsymbol{\Lambda}(\boldsymbol{x})\ddot{\boldsymbol{x}}_d + \boldsymbol{\mu}(\dot{\boldsymbol{x}}, \boldsymbol{x})\dot{\boldsymbol{x}}^m + \boldsymbol{\gamma}(\boldsymbol{x}) + \boldsymbol{K}_d(\boldsymbol{t})(\boldsymbol{x}_d - \boldsymbol{x}^m) + \boldsymbol{D}_d(\boldsymbol{t})(\dot{\boldsymbol{x}}_d - \dot{\boldsymbol{x}}^m) \tag{2.12}$$

When the end-effector reaches its goal position, it should stop at the goal and not have any velocity and acceleration. Hence, we set the desired end-effector velocity and acceleration as zero. Therefore, the task space variable impedance control (Equation (2.12)) changes to,

$$\boldsymbol{W}^c = \boldsymbol{\mu}(\dot{\boldsymbol{x}}, \boldsymbol{x})\dot{\boldsymbol{x}}^m + \boldsymbol{\gamma}(\boldsymbol{x}) + \boldsymbol{K}_d(t)(\boldsymbol{x}_d - \boldsymbol{x}^m) - \boldsymbol{D}_d(t)\dot{\boldsymbol{x}}^m \tag{2.13}$$

Our task space variable impedance control now generates the phantom force ($\boldsymbol{W}^c$) pulling on the end-effector. We can now formulate the admittance controller, which converts the phantom force into end-effector acceleration.

2.2 Admittance Controller for Object-Picking Task

Admittance control, like impedance control, is a control technique that provides a relationship between force, position, velocity, and acceleration. But unlike impedance control, admittance control provides motion to a robot when a force is applied by the environment on the robot arm. The force applied by the environment is modeled as a mass-spring-damper system, generating robot acceleration and resulting in motion (Refer to Figure 2.2).

add.png

Figure 2.2: External Force applied on a manipulator causing motion due to Admittance Controller.

Imagine pulling on a spring; when you apply force at the end of the spring, it displaces as a function of the force applied and its stiffness. Similarly, when an admittance control is deployed on a manipulator, the force applied on it generates motion of the arm as a function of the force applied and its stiffness ($\boldsymbol{K}_{ad}$) and damping ($\boldsymbol{D}_{ad}$) matrices. Therefore,

$$\boldsymbol{W} = \boldsymbol{M}_d \ddot{\boldsymbol{x}}^A + \boldsymbol{K}_{ad}(\boldsymbol{x}^m - \boldsymbol{x}_d) + \boldsymbol{D}_{ad} \dot{\boldsymbol{x}}^m \tag{2.14}$$

Where, $\boldsymbol{W}$ is the force acting on the robot arm (6×1), $\boldsymbol{M}_d$ is the desired inertia matrix (6×1), $\boldsymbol{K}_{ad}$ is the desired admittance stiffness matrix (6×6), and $\boldsymbol{D}_{ad}$ is the desired admittance damping matrix (6×6).

In the object-picking task, we want the robot to move to the object and lift it to the desired position. Here, we only know the desired position, and so we use a variable impedance controller to generate the force which the admittance controller uses, $\boldsymbol{W} = \boldsymbol{W}^c$, to calculate the end-effector acceleration ($\ddot{\boldsymbol{x}}^A$) guiding the robot toward the goal. Therefore,

$$\ddot{\boldsymbol{x}}^A = \boldsymbol{M}_d^{-1}(\boldsymbol{W}^c - \boldsymbol{K}_{ad}(\boldsymbol{x}^m - \boldsymbol{x}_d) - \boldsymbol{D}_{ad} \dot{\boldsymbol{x}}^m) \tag{2.15}$$

2.3 Twin-Delayed Deep Deterministic Policy Gradient (TD3)

As discussed in Section 2.1, the object-picking task requires a variable impedance control to generate the force necessary to lift objects with varying mass. Now that we have our variable impedance control (Equation (2.13)) and admittance control (Equation (2.15)), we can implement a deep reinforcement learning algorithm to learn the stiffness and damping parameters for variable impedance controller.

TD3, a successor to Deep Deterministic Policy Gradient (DDPG), is an off-policy algorithm widely used to solve continuous control problems. Although DDPG can solve continuous control problems with high performance, it can be sensitive to hyperparameters and other tuning parameters [19]. Both DDPG and TD3 learn Q-functions. Unlike DDPG, which can overestimate Q-values of the critic (value) network when built over time, leading to the agent being stuck at a local optimum [19], TD3 instead uses two Q-functions (Q_{ϕ_1} and Q_{ϕ_2}), hence the "twin", using the lower of the two Q-values to avoid overestimation and also delays the updates of the actor-network, hence the "delayed," which further reduces the possibility of overestimating the Q values. Another trick TD3 uses is the introduction of noise in the target action, preferring robust actions with higher values [19].

To understand the working of TD3 and its difference from DDPG, we must discuss the key features of TD3, i.e., target policy smoothing and clipped double-Q learning. Policy smoothing in TD3 refers to the smoothing of the Q-function of the target policy ($\mu_{\theta_{targ}}$) by adding clipped noise (ϵ), where $-c < \epsilon < c$ and $c \in \mathbb{N}$, to the target action ($a'(s')$) which is further clipped to fall under action limits ($a_{low} < a < a_{high}$). Policy smoothing helps avoid exploitation of actions with a high peak by the policy [19]. The target action is,

$$a'(s') = clip(\mu_{\theta_{targ}}(s') + clip(\epsilon, -c, c), a_{low}, a_{high}) \tag{2.16}$$

TD3 uses double-Q learning inspired by the Double Q-learning introduced by Van Hasselt, 2010, to select the Q value of the smaller critic networks. Therefore, the target value is,

$$y(r, s', d) = r + \gamma \min_{i=1,2} Q_{\phi_{i,targ}}(s', a'(s')) \tag{2.17}$$

The critic networks are then learned by regressing to the target value by using the mean-squared Bellman error (MSBE) function,

$$L(\phi_1, R) = \underset{(s,a,r,s',d)\sim R}{E}[(Q_{\phi_1}(s, a) - y(r, s', d)] \tag{2.18}$$

$$L(\phi_2, R) = \underset{(s,a,r,s',d) \sim R}{E}[(Q_{\phi_2}(s, a) - y(r, s', d)] \tag{2.19}$$

Where ϕ_i is the critic parameters, R is the transition tuple (s, a, r, s', d). d indicates whether state s' is the terminal state, a is the action performed at state s for which we get the reward r. Further, the policy learning is the same as in DDPG by maximizing Q_{ϕ_1}.

Chapter 3

Control Framework for Object-Picking Task

3.1 Manipulator Control Laws for Approaching and Lifting Phases

Now that we have introduced all three main components of our control framework, we can combine them (refer to Figure 3.1). This control framework works for both phases of the object-picking task. Note that though the framework is the same, the DRL algorithm needs to be trained separately for the two phases.

Referring to Figure 3.1, the only input to the framework is the desired end-effector position ($\boldsymbol{x}_d$). As discussed previously, the variable impedance controller with the DRL agent will derive the force ($\boldsymbol{W}^c$) necessary to move the end-effector (Equation (2.12)). The admittance controller will then convert the force into end-effector acceleration (Equation (2.15)). Since we use position-controlled UR5, we then convert the end-effector acceleration ($\ddot{\boldsymbol{x}}^A$) into end-effector position ($\boldsymbol{x}^c$) (Equation (3.3)). UR5 manipulator provides an controller.png

Figure 3.1: Control Framework Illustration.

interface where you command joint actuation values to move the arm. Since we know the end-effector position, using inverse kinematics (IK), we calculate the necessary joint actuation values ($\boldsymbol{q}^c$) and actuate the joints. Using the joint sensor measurements as feedback, which is converted to end-effector position ($\boldsymbol{x}^m$) and velocity ($\dot{\boldsymbol{x}}^m$) using forward kinematics and forward differential kinematics is compared with the desired position to vary the variable impedance force.

The approach phase is a more straightforward task where the only uncertainties are the object and end-effector locations, and the DRL training is relatively simpler. But for the lifting phase, where the object height when lifted is proportional to the force generated by the variable impedance controller and is inversely proportional to the unknown mass of the object, the training is much more complex. Here, the TD3 algorithm needs to observe the initial displacement of the object for the applied force to estimate the object weight and modulate the force generated by variable impedance control to not overshoot the goal or be unable to lift the object. By combining Equations (2.15) and (2.13) we get our control law,

$$\ddot{\boldsymbol{x}}^A = \boldsymbol{M}_d^{-1}(\boldsymbol{\mu}(\dot{\boldsymbol{x}}, \boldsymbol{x})\dot{\boldsymbol{x}}^m + \boldsymbol{\gamma}(\boldsymbol{x}) + \boldsymbol{K}_d(t)(\boldsymbol{x}_d - \boldsymbol{x}^m) \\ -\boldsymbol{D}_d(t)\dot{\boldsymbol{x}}^m - \boldsymbol{K}_{ad}(\boldsymbol{x}^m - \boldsymbol{x}_d) - \boldsymbol{D}_{ad}\dot{\boldsymbol{x}}^m) \tag{3.1}$$

Validation of this control framework is especially challenging when implemented on a position-controlled or velocity-controlled manipulator arm such as UR5. When lifting any object to a certain position with some velocity, UR5 applies the effort necessary to lift the object without providing any control over the applied effort. This can obscure the results of the control framework. To avoid this, we subtract the load of the object from the variable impedance force to mimic the behavior of reduced motion due to the weight of the object. This is only done for the lifting phase to mimic the behavior of a torque-controlled manipulator and is not required for manipulators that natively offer torque-control interfaces. Therefore, the control law for the lifting phase will change,

$$\ddot{\boldsymbol{x}}^A = \boldsymbol{M}_d^{-1}(\boldsymbol{\mu}(\dot{\boldsymbol{x}}, \boldsymbol{x})\dot{\boldsymbol{x}}^m + \boldsymbol{\gamma}(\boldsymbol{x}) + \boldsymbol{K}_d(t)(\boldsymbol{x}_d - \boldsymbol{x}^m) \\ -\boldsymbol{D}_d(t)\dot{\boldsymbol{x}}^m - \boldsymbol{K}_{ad}(\boldsymbol{x}^m - \boldsymbol{x}_d) - \boldsymbol{D}_{ad}\dot{\boldsymbol{x}}^m - M_o\boldsymbol{g}) \tag{3.2}$$

Where M_o is the object mass (kg) and $\boldsymbol{g}$ is the acceleration due to gravity ($1{\times}6$). The weight of the object can also induce a moment at the object and gripper contact point when the gripper is off-center to the object (Refer to Figure 3.2), but we can ignore that since we fix the orientation of the gripper and object which will be explained further in Section 3.2. As introduced, UR5 is either velocity or position-controlled, and since we have acceleration from the admittance controller, we need to convert it to commands acceptable by the UR5.

contact_moment.png

Figure 3.2: Moment generated due to Gripper grasping offset.

For the object-picking task where the objective is to reach a goal position, we convert admittance control acceleration ($\ddot{\boldsymbol{x}}^{A}$) into end-effector position ($\boldsymbol{x}^{c}$) (refer to Equation (3.3)). A position controller UR5 manipulator allows us to limit the motion of the arm within a set boundary, helping us avoid collisions with itself or the table. Using a position control makes it possible to clip the manipulator's position within a set boundary. Using kinematics equation we convert the admittance control acceleration $\ddot{\boldsymbol{x}}^{A}$ into end-effector position command $\boldsymbol{x}^c$,

$$\boldsymbol{x}^c(t) = \boldsymbol{x}^m(t) + \dot{\boldsymbol{x}}^m(t)t + \frac{1}{2}\,\ddot{\boldsymbol{x}}^A(t)\,t^2 \tag{3.3}$$

After obtaining the end-effector position, we use inverse kinematics to calculate joint angular position values ($\boldsymbol{q}^c$), which can then be commanded to the UR5 arm. Now that we have our control law for both the approach phase (Equation (3.1)) and the lifting phase (Equation (3.2)), we need to train the DRL agent for individual tasks, but before that, we need first to select an appropriate action space, observation space, and the reward function.

3.2 Simplification and Assumptions for Deep Reinforcement Learning

The complexity of the DRL task is heavily dependent on its action and observation space. Selecting an appropriate action and observation space size and shape is imperative in speeding up the learning process. The observation space for the object-picking task consists of the end-effector current and desired pose, both having 4×4 dimensions. Similarly, the action space is the stiffness and damping matrices of the variable impedance controller. Both stiffness and damping matrices are 6×6 matrices which, even when reduced to only selecting diagonal elements, reduces the action space to 1×12 array. We must shrink the observation and action space to simplify and speed up the learning.

In the object-picking task, where the object is a cube with a fixed shape, the gripper can grasp the object successfully by having its fingers parallel to the cube's face. To ensure that the object is grasped every time, we must fix the cube's and the gripper's orientation. The objective of the control framework is to reach the cube and lift it to the goal successfully and not find the appropriate grasping orientation. Selecting a fixed cube and gripper orientation can reduce the observation space to 1×6 array of the current and desired end-effector positions.

Since we have a fixed orientation of the object and the end-effector and do not want any unnecessary motion concerning the orientation of the object and end-effector, we can assume to have extremely high stiffness and damping for dimensions that correspond to the stiffness of the orientation axes. By doing so, we can eliminate the action space by half. Taking inspiration from [10], we can use a multiplier ($\xi \in \mathbb{N}$) to create a relationship between the stiffness and the damping matrices,

$$\boldsymbol{D}_d(t) = \xi \cdot \boldsymbol{K}_d(t) \tag{3.4}$$

We can further simplify the task by reducing the action space to just 3 dimensions. Now that we have defined our action and observation space, we need to create our reward function, which will guide our training for the object-picking task.

3.3 Reward Function

The object-picking task can be broken down into two phases: the approach phase and the lift phase. Both these phases can be formulated as a go-to-goal problem, with the only difference being whether the manipulator arm holds the object. The go-to-goal problem is attributed to the distance of the end-effector to the goal and can be formulated in the task space as,

$$\min_{\boldsymbol{u}(t)} \quad \boldsymbol{x}_d -^m \boldsymbol{x}(t) \tag{3.5}$$

$$s.t. \quad {}^{m}\boldsymbol{x}(t+1) = f({}^{m}\boldsymbol{x}(t), \boldsymbol{u}(t))$$

Where, $\boldsymbol{x}_d$ is the desired goal position, ${}^{m}\boldsymbol{x}(t)$ is the measured current end-effector position, $\boldsymbol{u}(t)$ is the action performed, and f is the unknown system dynamics. In the proposed control framework, the action $\boldsymbol{u}(t)$ is the stiffness matrix ($\boldsymbol{K}_d(t)$) selected by the TD3 algorithm.

Since the objective is to minimize the distance to the goal, the short-term reward function for the TD3 algorithm is set as,

$$r_s = \min_{\boldsymbol{K}_d(t)} \sum_{t=1}^{T} -100 \times \|\boldsymbol{x}_d -^{m} \boldsymbol{x}(t)\| \tag{3.6}$$

Whereas a high terminal positive reward, r_t, is given to the agent for successfully completing the task. Now that we have established the action space, the observation space, and the reward function, we can start our training for the approach and the lifting phases.

3.4 Training Using TD3

The pseudocode for the training with the TD3 algorithm is illustrated in Algorithm 1. We start with defining the hyperparameters that define the training scenario, such as maximum episodes, maximum steps, and batch size. In TD3, we also define the update interval, which is responsible for delaying network updates. Once the hyperparameters are set, we initialize the robot and task environment. They are responsible for performing the action the agent selects and generating observations and rewards for the agent to review for its next action decision. We then initialize the replay buffer, which holds the transition tuple containing state, action, reward, next state, and whether the state is a terminal state (done). We then initialize the actor, critic, and target neural networks containing predefined layers and nodes. Now that we have our training setup complete, we can start with the training.

DRL training is a repetitive task where every episode refers to one training scenario consisting of a predefined number of steps. We use a nested for-loop where the first loop runs for the maximum number of episodes defined in our hyperparameters, and the second loop is for the maximum allowable steps within an episode. The idea is to terminate an episode if the agent can't achieve its goal and restart the training with a new approach. At the beginning of every episode, we reset the robot and task environment and then perform the action the agent selects. Often it is a good idea to allow the agent to explore the environment and actions at the beginning of the training to have a better data set for learning. If we decide to allow the agent to explore for certain steps, the agent will select random actions from the action space and repeat until it has reached the maximum allowable exploration. Note that during the exploration phase, the episodes and step relation persist, and the environments will reset after every episode.

After the exploration phase, the agent selects actions using the neural network mapping, and we add some noise to the actions to make the learning more robust. After the robot environment executes the action, the task environment provides a reward with new observations. The transition tuple is then pushed to the replay buffer, which generates a table of data with a size equal to the defined batch size. After every step, the transition tuple is stored, and the episode reward is calculated. This continues till the task is complete or the maximum number of steps is reached.

Once filled up to the desired batch size, the replay buffer is used to train the networks after each episode. If the replay buffer is incomplete, the training moves on to the next episode without updating the networks. As discussed previously, TD3 uses a neat trick to avoid overestimation, known as delayed updates. The network models are stored and updated only after a few episodes. We also save the network models that can be loaded to produce the results of the trained model in the testing phase.

The testing phase is performed after the training is complete. Here, we load the saved trained model and run the model through multiple episodes of the task. The testing continues for a set number of episodes, and each episode runs till the task is complete.

Algorithm 1 TD3 Training and Testing Pseudocode

Set hyperparameters:
    max_episodes: Maximum number of training episodes
    max_steps: Maximum number of steps per episode
    batch_size: Number of experiences to consider from buffer
    explore_steps: Number of initial steps
    update_itr: Number of updates per step
    hidden_dim: Number of nodes in each hidden layer
    policy_target_update_interval: Interval for updating the policy and target networks
    explore_noise_scale: Scale for exploration noise
    eval_noise_scale: Scale for evaluation noise
Initialize robot and task environment
Initialize empty replay buffer ***R*** with Max Capacity
Initialize Q networks (critic) Q_{ϕ_1} and Q_{ϕ_2} and policy network (actor) π
Set target networks $Q'_{\phi_1} \leftarrow Q_{\phi_1}$, $Q'_{\phi_2} \leftarrow Q_{\phi_2}$, and $\pi' \leftarrow \pi$
if train is $True$ **then**
    for episodes in range(**max_episodes**) **do**
        Reset the Robot and Task Environment and get the current state
        Set Episode Reward to 0
        for step in range(**max_steps**) **do**
            if $frame\_idx$ is greater than explore_steps **then**
                Select action with exploration noise
            else
                Sample action from the action range
            end if
            Execute action and get the next_state, reward, done, and info from the Environment
            Push transition tuple (state, action, reward, next_state, done) to ***R***
            Replace state with next_state
            Add reward to Episode Reward
            Increase $frame\_idx$ by 1

Algorithm 1 continued $\cdots$

if len(R) is greater than **batch_size then**
for i in range(**update_itr**) **do**
Update the networks
end for
end if
if done **then**
Break
end if
end for
Append the Episode reward to the reward list
if episode is even and greater than 0 **then**
Save reward list
Save the Model
end if
end for
Save the Model
end if
if test is $True$ **then**
Load the trained model
for episodes in range(10) **do**
Reset the Robot and Task Environment and get the current state
Set Episode Reward to 0
Set done as $False$
while not done **do**
Select Action with exploration noise
Execute action and get the next_state, reward, done, and info from the Environment
Add reward to Episode Reward
replace the state with next_state
end while
end for
end if

Chapter 4

Simulations

After deriving our control law and designing the task as a DRL problem, we will now simulate and test the performance and validate our control framework. The chapter is separated into two parts; the first section explains the task setup in the Gazebo simulator and the task and robot-related parameters. Whereas the second section discusses the DRL setup and its parameters.

4.1 Simulation Setup and Parameters

Using Gazebo, a 3D robotics simulation package, we create our task environment (Refer to Figure 4.1). The environment consists of a UR5 robot arm equipped with a Robotiq 2f-85 gripper and the object to be picked. We restrict the task space of the robot arm within a bounded box, as shown in Figure 4.1 to avoid collisions with the table and with itself. The object in focus is a cube with unknown mass (M_o) and needs to be lifted by a UR5 manipulator arm.

In Section 3.1, we discuss we need two separate control laws for the approach and lift phases due to the lack of torque interface in the UR5 arm. In the lift phase, we subtract scenario.png

Figure 4.1: Object-picking scenario setup in Gazebo simulator.

the variable impedance force with the force due to the object's mass to mimic behavior similar to a torque-controlled robotic arm without the DRL agent knowing the object's mass. The training and controller validation is performed within the Gazebo simulator, allowing us to randomize the object mass in every training episode and making it possible to observe the random mass, which can then be subtracted from the phantom force ($\boldsymbol{W}^c$) generated by the variable impedance controller in the lifting phase. This allows us to reduce or increase (depending on the direction of the phantom force) the effect of the phantom force, resulting in a decreased acceleration output by the admittance controller and, hence, reduced positional or velocity control command.

In Section 3.2, we discuss fixing the orientation of the robot arm and the cube to reduce the observation and action spaces. The orientation of the robot arm can be fixed by keeping the rotation matrix of the homogeneous transformation constant. The configuration of choice is $\boldsymbol{q} = [0.0, -1.57, 1.57, -1.57, -1.57, 1.57]^T \quad rad$, where $\boldsymbol{q}$ is the join position value. This configuration can be seen in Figure 4.1 and allows the robot arm to move within the permitted workspace and provides ideal grasping. The permitted workspace is of volume $0.4 \times 0.44 \times 0.45 \ m^3$. The object mass can vary between 1 kg to 4 kg, which is a reasonable range as the maximum payload capacity for the UR5 arm is 5 kg.

UR5 manipulator joints can achieve the maximum velocity of 3.14 rad/s, which is higher than we desire. We limit the end-effector velocity to 1 m/s. Since we are using position control instead of velocity control, we limit the maximum end-effector velocity by limiting the maximum end-effector displacement of 0.2 m for a time period of 0.2 s. The maximum displacement and time period are selected based on observing the robot's behavior in the simulator.

4.2 Deep Reinforcement Learning Setup and Parameters

The TD3 algorithm, similar to other DRL techniques, requires us to set up the training hyperparameters, such as the maximum episodes, maximum steps in an episode, batch size, policy update interval, and exploration steps. We implement the TD3 algorithm inspired by the GitHub repository [15]. The policy update interval is the hyperparameter responsible for the delayed policy updates and is carried forward from the GitHub repository [15]. For both the approach and lift phases, we could train the DRL agent in 450 episodes with a maximum of 50 steps. The exploration steps allow us to select the number of steps at the beginning of the training the agent needs to explore. An initial exploration step of 300 with a batch size of 300 would give the agent enough experience to start learning. The hyperparameters for any DRL task can be extremely sensitive and require fine-tuning and intuition to set up. The action and observation spaces of the DRL task directly affect the complexity and the speed of the learning process. We discussed in Section 3.2 the technique to reduce the size of the action and observation spaces for the object-picking task. In this section, we will further discuss the action and observation spaces by selecting the appropriate range for our task.

The action space, i.e., $\boldsymbol{K}_d(t)$, is set to a maximum 600 N/m^2 and the multiplier, ξ, is set to 10 for the approach phase. Whereas for the lifting phase, $\boldsymbol{K}_d(t)$, is set to a maximum 1200 N/m^2. The increase in the stiffness parameter is due to the excess force required to lift the object in the lifting phase as compared to no object load in the approach phase. Also, as the end-effector reaches closer to its desired position, the force due to variable impedance control decreases significantly, requiring higher stiffness values to generate enough force to lift the object. Hence, the action space is,

Table 4.1: Action Space

Phases \ Actions	Low (N/m^2)			High (N/m^2)		
	x	y	z	x	y	z
Approach	-600	-600	-600	600	600	600
Lift	-1200	-1200	-1200	1200	1200	1200

The observation space in our task is an array of current and desired end-effector positions. We want the robot to move freely within its permitted workspace to increase the task's difficulty while keeping it safe from collisions. So the current end-effector position can be anywhere within the permitted workspace. In the approach phase, the desired end-effector position is the object's position in the world frame. The object is spawned randomly at different positions on the table within the permitted workspace. In the lifting phase, the desired end-effector position is the desired lifting position instead of the object position. Hence, the observation space for the object-picking task is,

Table 4.2: Observation Space

Phases \ Observations	Low (m)						High (m)					
	Current			Desired			Current			Desired		
	x	y	z	x	y	z	x	y	z	x	y	z
Approach	0.3	-0.22	0.5	0.4	-0.2	0.5	0.7	0.22	0.9	0.7	0.2	0.9
Lift	0.3	-0.22	0.5	0.3	-0.2	0.7	0.7	0.22	0.95	0.7	0.2	0.95

As discussed in Section 3.3, we provide a short-term reward, r_s, which is a function of distance to the goal at every time step. Whereas a high positive reward, 2000 pts, is given to the agent for successful completion of the task with an additional reward for high accuracy in $\boldsymbol{X}$, r_x, and $\boldsymbol{Y}$ axes, r_y. The task is said to be completed when the end-effector breaches a threshold distance $\boldsymbol{d_t}$. The task completion threshold is set to be 0.025 m for the approaching phase and 0.035 m for the lifting phase. The lower threshold distance in the approach phase allows the gripper to move in close enough for successful grasping. The additional rewards for $\boldsymbol{X}$ and $\boldsymbol{Y}$ accuracy make sure the end-effector gripper is centered on the object in the approach phase for a good object grasp. We avoid providing the same accuracy reward in the $\boldsymbol{Z}$ axis since the gripper extends when grasping and can collide with the table (refer to Figure 4.2). We can make it so that the extended gripper fingers' positions are considered, but we will then need to increase the threshold distance by the equivalent increment so that the gripper fingers can grasp the object, leading to the same training setup.

Figure 4.2: Gripper position offset in z-axis.

The simulation and DRL setup is complete and we can train the agent for the two sub-tasks, approach and lift, and validate the control framework. The main parameter to observe during the training and testing of the control framework, is the distance of the end-effector to its desired position.

Chapter 5

Results

This thesis presented a novel control framework that employed a task-space variable impedance controller learned using the TD3 algorithm and a task-space admittance controller to convert the phantom force generated by the variable impedance controller into end-effector acceleration. The motive was to achieve a human-like object-picking behavior, which varied the force applied by the robot to pick an object of unknown mass. We separated the object-picking task into two phases, the approaching and the lifting phase, and derived control law and the DRL training scenarios for both.

This chapter discusses the results of each phase of the object-picking task. The performance for both phases is measured with the end-effector's ability to reach the desired position using the control law specific to the scenario. The threshold distance required to be met by the end-effector can be reduced to improve the robot's accuracy. Still, our motive is to validate the control framework and decreasing the threshold distance would require longer training times and a high-performance workstation.

5.1 Approach Phase

In the approach phase, the object position was the desired end-effector position for the DRL task. The short-term reward to the agent was the distance to the object position with high positive reward when reaching the threshold distance. We also provided additional rewards for high accuracy in X and Y directions. We measure the performance of DRL training and the control framework by observing the difference in object position and end-effector position and the reward it gets after each episode. If the control law and the reward functions are effective, we should see a reduction in the distance to the goal and an increase in reward values as the training progresses.

Figures 5.1, 5.2, and 5.3 show the difference between the object and the end-effector distances in each of the axes. The plots show a confidence interval (95%), light blue shaded region, and mean. We can observe that as the training progresses, we see a reduction in the difference between the object and the end-effector at the end of each episode and a decrease in the confidence interval, where the majority of the learning can be observed within the first 100 episodes.

approach_diff_x.png

Figure 5.1: Difference in x in Approach Phase training.

approach_diff_y.png

Figure 5.2: Difference in y in Approach Phase training.

approach_diff_z.png

Figure 5.3: Difference in z in Approach Phase training.

Figure 5.4 showcases the reward achieved by the DRL agent at the end of each episode with a confidence interval (95%) and mean. We observe a similar trend, as seen in the distance difference plot, where the majority of the learning can be observed in the first 100 episodes and by episode 300, the agent has completely learned the policy.

approach_ep_reward.png

Figure 5.4: Episode Reward Confidence Plot for Approach Phase

5.2 Lifting Phase

In the lifting phase, the goal position was the desired end-effector position for the DRL task and same as the approach phase, the short-term reward to the agent is the distance to the goal position with high positive reward when reaching the threshold distance. In the lifting phase, we use a higher threshold distance as compared to the approach phase to speed up the learning process, and the effects of this can be seen in Figures 5.5 and 5.7 where the distance to goal is higher compared to the approach phase. We measure the performance of DRL training and the control framework by observing the difference in goal and end-effector positions and the reward it gets after each episode. If the control law and the reward functions are effective, we should see a reduction in distance to the goal and an increase in reward values as the training progresses. We will also observe the result of the trained model in Figure 5.9.

Figures 5.5, 5.6, and 5.7 show the difference between the goal and the end-effector distances in each of the axes. The plots show a confidence interval (95%), light blue shaded region, and mean. As the training progresses, we observe a reduced difference between the object and the end-effector at the end of each episode and a decreasing confidence interval where most of the learning can be observed within the first 100 episodes.

lift_diff_x.png

Figure 5.5: Difference in x in lift phase training.

lift_diff_y.png

Figure 5.6: Difference in y in lift phase training.

lift_diff_z.png

Figure 5.7: Difference in z in lift phase training.

Unlike the approach phase, where we rapidly increase the episode reward and reach a maximum reward higher than 2000 pts, the lifting phase requires higher episodes to reach its maximum reward (Refer to Figure 5.8). The maximum reward in the lift phase is lower than in the approach phase. This is because it takes higher steps to reach the goal. The confidence interval in the lifting phase is wider than in the approaching phase.

lift_ep_reward.png

Figure 5.8: Episode Reward confidence plot for lift phase.

Figure 5.9 illustrates the trained DRL model for the lifting phase. Here, we deploy the trained model for five runs, and in every run, the object position is randomized. The different lines depict the object's position. We can observe that the agent can use the proposed control law to lift the object to the exact goal location for multiple runs.

lift_test.png

Figure 5.9: Lift phase trained model runs.

5.3 Comparison Study

To further display the merit of the proposed framework in lifting an object of unknown mass, we simulate the same scenario, but instead of using a variable impedance controller to generate the phantom force, we select two controllers, fixed impedance and variable PD controllers, for our comparison study. The framework remains the same, but only the phantom force-generating variable impedance controller will be replaced by either a fixed impedance controller or a variable PD controller. The two controllers are selected to directly compare the efficacy of variable impedance in adapting to the unknown object mass with the two popular controllers. Both fixed impedance and variable PD controllers will be trained using the TD3 algorithm with the same hyperparameters for ideal comparison.

Upon conducting the training, we observed that training for fixed impedance and variable PD controllers would end without achieving the desired training episodes and without learning an optimal policy to pick an object of mass between 1 and 4 kg. Both fixed impedance and variable PD controllers couldn't adapt to the unknown object mass and required a higher action space range and reduced object mass variance to reach the desired learning episodes. For the fixed impedance, the object mass was reduced to vary between 1 to 2 kg, and for variable PD, the object mass was reduced to vary between 1 to 2.5 kg in contrast to the variable impedance controller, which completed the training for 1 to 4 kg of object mass range.

We then trained the variable impedance controller for a reduced object mass range, 1 to 2 kg, to better compare the three controllers. Starting with the fixed impedance controller (refer to Figure 5.10), the optimal policy learned by the agent saturates the episode reward of just over 1000 pts for the lifting phase.

fixd_imp_episode_reward.png

Figure 5.10: Episode reward confidence plot for fixed impedance controller during lift phase.

Similarly, in Figure 5.11, we observe the variable PD controller starts learning, and the episode reward increases as the training continues. Still, it can only reach a maximum of 500 pts episode reward by the end of the training period. With this, we have our benchmark to compare our proposed framework to.

pd_lift_episode_reward.png

Figure 5.11: Episode reward confidence plot for variable PD controller during lift phase.

As discussed at the beginning of this section, we train our proposed framework with a reduced object mass range to compare to the benchmark set by fixed impedance and variable PD controller. Figure 5.12 shows the proposed framework's training simulation with reduced object mass variance, 1 to 2 kg. The framework can quickly adapt to the varying object mass (within 100 episodes), learn an optimal policy and reach a maximum reward of about 2000 pts by the end of the training. This showcases the superiority of the variable impedance controller in our specific task. The proposed framework can reduce the training speed and achieve a higher reward per episode than fixed impedance and variable PD controllers.

vimp_redcd_weight_episode_reward_2.png

Figure 5.12: Episode reward plot (smoothed) for the proposed framework with reduced object mass range.

Chapter 6

Conclusion

In this thesis, we proposed a novel framework to lift an object with an unknown mass. The idea is to mimic human-like object-picking behavior by applying force based on the realized object mass. We deploy three main techniques for this: variable impedance control, TD3 algorithm, and admittance control. The manipulator of choice is a UR5 manipulator, and the object to be picked is a cube of 1 to 4 kg mass. The object-picking task is broken into two phases: approaching and lifting.

Variable impedance control generates force as a function of distance to the goal and the stiffness and damping matrices. Since the distance to the goal for any object mass can be the same, resulting in the same force for different object masses, we use the stiffness and damping matrix to modulate the force generated. As the object mass is unknown, the stiffness and damping matrices must be varied to generate appropriate phantom force to lift the object at every episode. To realize the object's mass and vary the phantom force to be able to lift that object requires machine learning.

Deep reinforcement learning algorithms are especially effective in such model-free tasks. We use twin-delayed deep deterministic policy gradient (TD3), an off-policy DRL algorithm. We design our task as a DRL problem and tune the hyperparameters to achieve the desired learning. Now that we have the generated force required to lift the object, we need to convert the force into communicable control for the UR5 arm. UR5 arm only allows us to control the joint position and velocity and doesn't provide us with any control over its joint torque. This limitation requires us to convert the phantom force into a joint position or velocity.

Admittance control is a popular choice to convert force applied on a robot's end-effector into motion. The idea is to use the force generated by the variable impedance controller and TD3 as an external force pulling (phantom force) on the end-effector to the desired position. The admittance controller converts the force into end-effector acceleration. As UR5 is either a velocity-controlled or a position-controlled robot, we need to convert end-effector acceleration into either end-effector velocity or position using the kinematics equation, which can then be converted into joint actuation values using inverse kinematics. We opt for position control as it allows us to restrict the motion of the robot arm within a permitted workspace.

Validating our control framework on a position-controlled UR5 is impossible without adjusting the control law. When using position control, UR5 applies the effort necessary to reach the position without providing any interface to control the effort. This would mean that no matter what the object's mass is, UR5 would reach the desired position. For this, we deduct the force due to the object's weight from the control law in the lifting phase, reducing the end-effector acceleration and mimicking a similar effect to what would be observed in a torque-controlled manipulator.

After deriving our control law and parameters for the DRL problem, we simulate and train the agent in Gazebo and PyTorch. The training data is analyzed, and the lifting phase is tested. We observe successful training of the model in both the approaching and lifting phases. The distance to the goal decreases with every training episode while the rewards increase. Further, we perform a comparison study wherein the proposed framework is pitted against a fixed impedance and a variable PD controller. Both fixed impedance and variable controller are integrated into the proposed framework by replacing the variable impedance controller to generate the phantom force. The outcome of the comparison study showcases the superiority of the variable impedance controller over the other two controllers by learning an optimal policy quicker and gaining higher reward per episode. Future research will focus on a more in-depth analysis of the control framework by assessing the force and displacement of the end-effector and deploying the trained model on a physical UR5 robot.

Appendices

Appendix A TD3 Script

```
#!/usr/bin/env python

import math
import random

import gymnasium as gym
import numpy as np

# Torch imports
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.distributions import Normal
from torch.utils.tensorboard import SummaryWriter

from IPython.display import clear_output
import matplotlib.pyplot as plt
from matplotlib import animation
from IPython.display import display

# Robot and task space import
from robo_env import ROBO_ENV
from ur5_reaching import UR5_REACHING
from ur_imp_lift import UR_IMP_LIFT
from ur_imp_reach import UR_IMP_REACH
from ur_pd_reach import UR_PD_REACH
from ur_pd_lift import UR_PD_LIFT

```

```
import argparse
import time

# Comment out seeds and only keep 1 at a time
torch.manual_seed(1234)  #Reproducibility
torch.manual_seed(1000)
torch.manual_seed(900)
torch.manual_seed(800)
torch.manual_seed(700)
torch.manual_seed(600)

GPU = True
device_idx = 0
if GPU:
    device = torch.device("cuda:" + str(device_idx) if torch.cuda.
   is_available() else "cpu")
else:
    device = torch.device("cpu")
print(device)


class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
```

```
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = int((self.position + 1) % self.capacity)  # as a ring buffer

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = map(np.stack, zip(*batch)) # stack for each element
        '''
        the * serves as unpack: sum(a,b) <=> batch=(a,b), sum(*batch) ;
        zip: a=[1,2], b=[2,3], zip(a,b) => [(1, 2), (2, 3)] ;
        the map serves as mapping the function on each list element: map(square, [2,3]) => [4,9] ;
        np.stack((1,2)) => array([1, 2])
        '''
        return state, action, reward, next_state, done

    def __len__(self):
        return len(self.buffer)

class NormalizedActions(gym.ActionWrapper):
    def _action(self, action):
        low  = self.action_space.low
        high = self.action_space.high

        action = low + (action + 1.0) * 0.5 * (high - low)
        action = np.clip(action, low, high)

        return action

```

```
    def _reverse_action(self, action):
        low  = self.action_space.low
        high = self.action_space.high

        action = 2 * (action - low) / (high - low) - 1
        action = np.clip(action, low, high)

        return action


class ValueNetwork(nn.Module):
    def __init__(self, state_dim, hidden_dim, init_w=3e-3):
        super(ValueNetwork, self).__init__()

        self.linear1 = nn.Linear(state_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, hidden_dim)
        self.linear4 = nn.Linear(hidden_dim, 1)
        # weights initialization
        self.linear4.weight.data.uniform_(-init_w, init_w)
        self.linear4.bias.data.uniform_(-init_w, init_w)

    def forward(self, state):
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear3(x))
        x = self.linear4(x)
        return x


class QNetwork(nn.Module):
```

```
    def __init__(self, num_inputs, num_actions, hidden_size, init_w=3e-3):
        super(QNetwork, self).__init__()

        self.linear1 = nn.Linear(num_inputs + num_actions, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, hidden_size)
        self.linear4 = nn.Linear(hidden_size, 1)

        self.linear4.weight.data.uniform_(-init_w, init_w)
        self.linear4.bias.data.uniform_(-init_w, init_w)

    def forward(self, state, action):
        x = torch.cat([state, action], 1) # the dim 0 is number of samples
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear3(x))
        x = self.linear4(x)
        return x


class PolicyNetwork(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, action_range=1., init_w=3e-3, log_std_min=-20, log_std_max=2):
        super(PolicyNetwork, self).__init__()

        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.linear1 = nn.Linear(num_inputs, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
```

```
        self.linear3 = nn.Linear(hidden_size, hidden_size)
        self.linear4 = nn.Linear(hidden_size, hidden_size)

        self.mean_linear = nn.Linear(hidden_size, num_actions)
        self.mean_linear.weight.data.uniform_(-init_w, init_w)
        self.mean_linear.bias.data.uniform_(-init_w, init_w)

        self.log_std_linear = nn.Linear(hidden_size, num_actions)
        self.log_std_linear.weight.data.uniform_(-init_w, init_w)
        self.log_std_linear.bias.data.uniform_(-init_w, init_w)

        self.action_range = action_range.detach().cpu()
        self.num_actions = num_actions


    def forward(self, state):
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = F.relu(self.linear3(x))
        x = F.relu(self.linear4(x))

        mean  = F.tanh(self.mean_linear(x))


        log_std = self.log_std_linear(x)
        log_std = torch.clamp(log_std, self.log_std_min, self.log_std_max)

        return mean, log_std

```

```
    def evaluate(self, state, deterministic, eval_noise_scale, epsilon=1e-6):
        '''
        generate action with state as input wrt the policy network, for calculating gradients
        '''
        mean, log_std = self.forward(state)
        mean = mean.cpu()
        std = log_std.exp() # no clip in evaluation, clip affects gradients flow

        normal = Normal(0, 1)
        z      = normal.sample()
        action_0 = torch.tanh(mean.to(device) + std*z.to(device)) # TanhNormal distribution as actions; reparameterization trick
        action_range = self.action_range.to(device)
        action = action_range*mean.to(device) if deterministic else action_range*action_0
        log_prob = Normal(mean.cpu(), std.cpu()).log_prob(mean.cpu()+std.cpu()*z.cpu()) - torch.log(1. - action_0.pow(2).cpu() + epsilon) -  np.log(action_range.cpu())

        log_prob = log_prob.sum(dim=1, keepdim=True)
        ''' add noise '''
        eval_noise_clip = 2*eval_noise_scale
        noise = normal.sample(action.shape) * eval_noise_scale
        noise = torch.clamp(noise, -eval_noise_clip, eval_noise_clip)
        action = action + noise.to(device)

        return action, log_prob, z, mean, log_std


```

```
    def get_action(self, state, deterministic, explore_noise_scale):
        '''
        generate action for interaction with env
        '''
        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        mean, log_std = self.forward(state)
        std = log_std.exp()

        normal = Normal(0, 1)
        z      = normal.sample().to(device)

        action = mean.detach().cpu().numpy()[0] if deterministic else torch.tanh(mean + std*z).detach().cpu().numpy()[0]

        ''' add noise '''
        noise = normal.sample(action.shape) * explore_noise_scale
        print('\nNoise: ', noise)
        action = self.action_range*action + noise.numpy()

        return action


    def sample_action(self,):
        a=torch.FloatTensor(self.num_actions).uniform_(-1, 1)
        return self.action_range*a.numpy()


class TD3_Trainer():
    def __init__(self, replay_buffer, hidden_dim, action_range, policy_target_update_interval=1):
        self.replay_buffer = replay_buffer
```

```


        self.q_net1 = QNetwork(state_dim, action_dim, hidden_dim).to(device)
        self.q_net2 = QNetwork(state_dim, action_dim, hidden_dim).to(device)
        self.target_q_net1 = QNetwork(state_dim, action_dim, hidden_dim).to(device)
        self.target_q_net2 = QNetwork(state_dim, action_dim, hidden_dim).to(device)
        self.policy_net = PolicyNetwork(state_dim, action_dim, hidden_dim, action_range).to(device)
        self.target_policy_net = PolicyNetwork(state_dim, action_dim, hidden_dim, action_range).to(device)
        print('Q Network (1,2): ', self.q_net1)
        print('Policy Network: ', self.policy_net)

        self.target_q_net1 = self.target_ini(self.q_net1, self.target_q_net1)
        self.target_q_net2 = self.target_ini(self.q_net2, self.target_q_net2)
        self.target_policy_net = self.target_ini(self.policy_net, self.target_policy_net)


        q_lr = 3e-5#3e-4
        policy_lr = 3e-5#3e-4
        self.update_cnt = 0
        self.policy_target_update_interval = policy_target_update_interval

```

```
        self.q_optimizer1 = optim.Adam(self.q_net1.parameters(), lr=q_lr)
        self.q_optimizer2 = optim.Adam(self.q_net2.parameters(), lr=q_lr)
        self.policy_optimizer = optim.Adam(self.policy_net.parameters(), lr=policy_lr)

    def target_ini(self, net, target_net):
        for target_param, param in zip(target_net.parameters(), net.parameters()):
            target_param.data.copy_(param.data)
        return target_net

    def target_soft_update(self, net, target_net, soft_tau):
    # Soft update the target net
        for target_param, param in zip(target_net.parameters(), net.parameters()):
            target_param.data.copy_(  # copy data value into target parameters
                target_param.data * (1.0 - soft_tau) + param.data * soft_tau
            )

        return target_net

    def update(self, batch_size, deterministic, eval_noise_scale, reward_scale=10., gamma=0.9,soft_tau=1e-2):
        state, action, reward, next_state, done = self.replay_buffer.sample(batch_size)
        # print('sample:', state, action, reward, done)

```

```
        state      = torch.FloatTensor(state).to(device)
        next_state = torch.FloatTensor(next_state).to(device)
        action     = torch.FloatTensor(action).to(device)
        reward     = torch.FloatTensor(reward).unsqueeze(1).to(device)  # reward is single value, unsqueeze() to add one dim to be [reward] at the sample dim;
        done       = torch.FloatTensor(np.float32(done)).unsqueeze(1).to(device)

        predicted_q_value1 = self.q_net1(state, action)
        predicted_q_value2 = self.q_net2(state, action)
        new_action, log_prob, z, mean, log_std = self.policy_net.evaluate(state, deterministic, eval_noise_scale=0.0)  # no noise, deterministic policy gradients
        new_next_action, _, _, _, _ = self.target_policy_net.evaluate(next_state, deterministic, eval_noise_scale=eval_noise_scale) # clipped normal noise

        reward = reward_scale * (reward - reward.mean(dim=0)) / (reward.std(dim=0) + 1e-6) # normalize with batch mean and std; plus a small number to prevent numerical problem

    # Training Q Function
        target_q_min = torch.min(self.target_q_net1(next_state, new_next_action),self.target_q_net2(next_state, new_next_action))

        target_q_value = reward + (1 - done) * gamma * target_q_min # if done==1, only reward

        q_value_loss1 = ((predicted_q_value1 - target_q_value.detach())**2).mean()  # detach:  no gradients for the variable
        q_value_loss2 = ((predicted_q_value2 - target_q_value.detach())**2).mean()
```

```python
        self.q_optimizer1.zero_grad()
        q_value_loss1.backward()
        self.q_optimizer1.step()
        self.q_optimizer2.zero_grad()
        q_value_loss2.backward()
        self.q_optimizer2.step()

        if self.update_cnt%self.policy_target_update_interval==0:
        # This is the **Delayed** update of policy and all targets.
        # Training Policy Function
            ''' implementation 1 '''
            ''' predicted_new_q_value = torch.min(self.q_net1(state, new_action),self.q_net2(state, new_action)) '''
            ''' implementation 2 '''
            predicted_new_q_value = self.q_net1(state, new_action)

            policy_loss = - predicted_new_q_value.mean()

            self.policy_optimizer.zero_grad()
            policy_loss.backward()
            self.policy_optimizer.step()

        # Soft update the target nets
            self.target_q_net1=self.target_soft_update(self.q_net1, self.target_q_net1, soft_tau)
            self.target_q_net2=self.target_soft_update(self.q_net2, self.target_q_net2, soft_tau)
            self.target_policy_net=self.target_soft_update(self.policy_net, self.target_policy_net, soft_tau)

        self.update_cnt+=1
```

```

        return predicted_q_value1.mean()

    def save_model(self, path):
        torch.save(self.q_net1.state_dict(), path+'_q1')
        torch.save(self.q_net2.state_dict(), path+'_q2')
        torch.save(self.policy_net.state_dict(), path+'_policy')

    def load_model(self, path):
        self.q_net1.load_state_dict(torch.load(path+'_q1'))
        self.q_net2.load_state_dict(torch.load(path+'_q2'))
        self.policy_net.load_state_dict(torch.load(path+'_policy'))
        self.q_net1.eval()
        self.q_net2.eval()
        self.policy_net.eval()

def plot(rewards):
    clear_output(True)
    plt.figure(figsize=(20,5))
    plt.plot(rewards)
    plt.savefig('td3.png')
    # plt.show()

# Only keep the env in focus, comment out rest
env = ROBO_ENV()
env = UR_IMP_LIFT()
env = UR_PD_REACH()
env = UR_PD_LIFT()
env = UR_IMP_REACH()
action_dim = env.action_space.shape[0]
state_dim = env.observation_space.shape[0]
```

```python
action_range = env.action_space.high
action_range = torch.tensor(action_range, dtype = torch.float32, device
    = device)#torch.device('cpu'))

replay_buffer_size = 5e5
replay_buffer = ReplayBuffer(replay_buffer_size)


# hyper-parameters for RL training
max_episodes  = 450
max_steps   = 50 #20
frame_idx   = 0
batch_size  = 300#150
explore_steps = 300  # for random action sampling in the beginning of training
update_itr = 1
hidden_dim = 256#512
policy_target_update_interval = 3 # delayed update:policy and target networks
DETERMINISTIC=True  # DDPG: deterministic policy gradient
explore_noise_scale = 0.1
eval_noise_scale = 0.1
reward_scale = 1.
rewards     = []
# Check model path before every run
model_path = './model/td3_imp_lift_redcd_weight'

td3_trainer=TD3_Trainer(replay_buffer, hidden_dim=hidden_dim,
    policy_target_update_interval=policy_target_update_interval,
    action_range=action_range )

if __name__ == '__main__':

```

```python
    # train = False
    train = True
    if train:

        writer = SummaryWriter(comment="TD3_IMP_lift_redcd_weight")
        episode_reward = 0
        rewards = []
        total_timesteps = 0

        # training loop
        for eps in range(max_episodes):

            state =  env.reset()
            episode_reward = 0

            for step in range(max_steps):

                if frame_idx > explore_steps:
                    action = td3_trainer.policy_net.get_action(state,
deterministic = DETERMINISTIC, explore_noise_scale=
explore_noise_scale)
                else:
                    action = td3_trainer.policy_net.sample_action()

                print("\nEpisode: ",eps,"| Step: ", step)
                next_state, reward, done, info = env.step(action)
                replay_buffer.push(state, action, reward, next_state,
done)

                state = next_state
                episode_reward += reward
```

```
            frame_idx += 1

            if len(replay_buffer) > batch_size:
                for i in range(update_itr):
                    _=td3_trainer.update(batch_size, deterministic=DETERMINISTIC, eval_noise_scale=eval_noise_scale, reward_scale=reward_scale)

            total_timesteps += 1
            writer.add_scalar("reward_step", reward, total_timesteps)
            if done:
                break

        rewards.append(episode_reward)
        avg_reward = np.mean(rewards[-100:])
        print("\nAvg_reward = ", avg_reward)
        writer.add_scalar("avg_reward", avg_reward, total_timesteps)
        writer.add_scalar("episode_reward", episode_reward, eps)

        writer.add_scalar("Difference in x", info[0], eps)
        writer.add_scalar("Difference in y", info[1], eps)
        writer.add_scalar("Difference in z", info[2], eps)

        if eps % 2 == 0 and eps>0:
            np.save('rewards_td3', rewards)
            td3_trainer.save_model(model_path)

        print('Episode: ', eps, '| Episode Reward: ', episode_reward)

```

```python
        td3_trainer.save_model(model_path)

    # test = True
    # test = False
    # if test:
    if not train:
        td3_trainer.load_model(model_path)
        for eps in range(10):

            state =  env.reset()
            episode_reward = 0
            done = False

            while not done:
                action = td3_trainer.policy_net.get_action(state, deterministic = DETERMINISTIC, explore_noise_scale=0.0)
                next_state, reward, done, _ = env.step(action)

                episode_reward += reward
                state=next_state



            print('Episode: ', eps, '| Episode Reward: ', episode_reward)
```

Listing 1: TD3 Python Code

Appendix B Variable Impedance Reaching Environment

```
#!/usr/bin/env python

# Gazebo Imports
import rospy
import rospkg
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState, GetModelState, GetLinkState
import control_msgs.msg
import actionlib
from trajectory_msgs.msg import *
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint
from geometry_msgs.msg import WrenchStamped
from std_srvs.srv import Empty

import numpy as np
import gymnasium as gym
import sys
import torch
import time

# Robotics toolbox -python imports for kinematics and dynamics of ur5
import roboticstoolbox as rtb
from spatialmath import SE3

class UR_IMP_REACH():

    def __init__(self):
```

```python

        rospy.init_node('ROBO_ENV', anonymous = True) # Initializing node

        self.jointstate = JointState()
        self.modelstate = ModelState()
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point = JointTrajectoryPoint()

        self.cube_name = 'cube1'
        self.cube_relative_entity_name = 'link'
        self.link_name = 'robot::left_inner_finger'

        self.robot = rtb.models.UR5() # Load UR5
        self.robot_dh = rtb.models.DH.UR5()

        # Gazebo Services
        self.model_coordinates = rospy.ServiceProxy('/gazebo/get_model_state', GetModelState)
        self.link_coordinates = rospy.ServiceProxy('/gazebo/get_link_state', GetLinkState)
        self.set_state = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)
        self.unpause = rospy.ServiceProxy('/gazebo/unpause_physics', Empty)
        self.pause = rospy.ServiceProxy('/gazebo/pause_physics', Empty)

        # Publisher and Subscriber
```

```
        self.ur_cmd = rospy.Publisher('/arm_controller/command', JointTrajectory, queue_size = 1)
        self.ur_jointstate = rospy.Subscriber('/joint_states', JointState, self.ur5_joint_callback)
        self.gripper_client = actionlib.SimpleActionClient('/gripper_controller/gripper_cmd', control_msgs.msg.GripperCommandAction)
        self.ft_sensor = rospy.Subscriber('/ft_sensor/raw', WrenchStamped, self.ft_sensor_callback)
        self.goal = control_msgs.msg.GripperCommandGoal()

        # Limits of end-effector position
        self.max = np.array([0.60, 0.22, 0.40, 0, 0, 0])#30])
        self.min = np.array([0.29, -0.22, 0.2, 0, 0, 0])#188])
        self.max_x = torch.tensor(self.max, dtype = torch.float32, device = torch.device("cpu"))
        self.min_x = torch.tensor(self.min, dtype = torch.float32, device = torch.device("cpu"))

        # Action space :  x direction,y direction,z direction:  task space
        self.action_space = gym.spaces.Box(low = np.array([-6,-6,-6]), high = np.array([6,6,6]), dtype= np.float32)

        self.max_action = self.action_space.high
        self.min_action = self.action_space.low

        # Observation Space = [x,y,z,cube.x,cube.y,cube.z]
        self.observation_space = gym.spaces.Box(low = np.array([30, -25, 20, 40, -15, 0]), high = np.array([70, 25, 35, 50, 15, 60]), dtype= np.float32)

```

```
        self.cuda0 = torch.device('cuda:0')

        self.reward = 0
        self.prev_reward = 0
        self.prev_distToGoal = 0
        self.distToGoal = 0
        self.done_counter = 0
        self.eps = 0.75
        self.Ka = 1*np.identity(6)
        self.Da = self.eps*self.Ka
        self.Md_a = 3*np.identity(6)
        self.t = 0.5

        # Desired Velocity and Acceleration
        self.xdot_d = np.zeros(6,).reshape((-1,1))
        self.xddot_d = np.zeros(6,).reshape((-1,1))

    def ur5_joint_callback(self, data):

        self.jointstate = data

    def ft_sensor_callback(self,data):

        self.ft_data = data

    def get_observation(self):

        self.q0 = self.jointstate.position

        # Cube Coordinates
```

```
        self.inner_finger_coord = self.link_coordinates(self.link_name, 'world')
        self.tcp_x = self.inner_finger_coord.link_state.pose.position.x - 0.0681975
        self.tcp_y = self.inner_finger_coord.link_state.pose.position.y
        self.tcp_z = self.inner_finger_coord.link_state.pose.position.z - 0.066 - 0.435
        self.tcp_coord = np.array([100*self.tcp_x, 100*self.tcp_y, 100*self.tcp_z])
        print("\nTCP Coordinates: ", self.tcp_coord)

        # Creating observation array
        self.obs = np.array([])
        self.obs = np.append(self.obs, self.tcp_coord)
        self.obs = np.append(self.obs, self.x_goal)

        return self.obs


    def reset(self):

        self.q_cmd1 = JointTrajectory()
        self.q_cmd2 = JointTrajectory()
        self.q_cmd1.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.q_cmd2.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
```

```
        self.point1 = JointTrajectoryPoint()
        self.point2 = JointTrajectoryPoint()

        self.q = [0.0,-1.57,1.57,-1.57,-1.57,1.57]

        # UR5 reset position
        self.q_dot_cmd = [0.0,0.0,0.0,0.0,0.0,0.0]
        self.og_Te = np.array(self.robot.fkine(np.array
    ([0.0,-1.57,1.57,-1.57,-1.57,1.57])))
        self.sol1 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q)
        self.point1.positions = self.sol1.q
        self.point1.velocities = -3*self.q_dot_cmd
        self.point1.time_from_start = rospy.Duration(1)
        self.q_cmd1.points.append(self.point1)
        # self.unpause()
        # time.sleep(0.5)
        # time.sleep(1.5)

        # Randomize UR5 gripper x and y location
        self.ur_x = np.random.uniform(0.30,0.59)
        self.ur_y = np.random.uniform(-0.15,0.15)
        self.og_Te[0][3] = self.ur_x
        self.og_Te[1][3] = self.ur_y
        self.sol2 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.sol1.
    q)
        self.point2.positions = self.sol2.q

        # Publish UR5 velocity and position
        self.unpause()
        self.point2.velocities = -3*self.q_dot_cmd
        self.point2.time_from_start = rospy.Duration(2)
```

```
        self.q_cmd1.points.append(self.point2)
        self.ur_cmd.publish(self.q_cmd1)
        time.sleep(0.5)
        # time.sleep(3)

        # Publish gripper as open and set gripper status = 0
        self.gripper_status = 0
        self.gripper_client.wait_for_server()
        self.goal.command.position = self.gripper_status
        self.goal.command.max_effort = -1.0 # Do not limit the effort
        self.gripper_client.send_goal(self.goal)
        self.gripper_client.wait_for_result()

        # Randomize x and y location of cube
        self.cube_x = np.random.uniform(0.4,0.6)
        self.cube_y = np.random.uniform(-0.15,0.15)

        # Cube reset position
        self.modelstate.model_name = 'cube1'
        self.modelstate.pose.position.x = self.cube_x #0.4
        self.modelstate.pose.position.y = self.cube_y #-0.1
        self.modelstate.pose.position.z = 0.6
        self.modelstate.pose.orientation.x = 0
        self.modelstate.pose.orientation.y = 0
        self.modelstate.pose.orientation.z = 0
        self.modelstate.pose.orientation.w = 0
        rospy.wait_for_service('/gazebo/set_model_state')

        try:
            self.resp = self.set_state(self.modelstate)
            # time.sleep(0.3)
```

```
            time.sleep(0.5)

        except rospy.ServiceException as e:
            print ("Service call failed: %s" % e)

        self.cube_coord = self.link_coordinates('cube1::link', 'world')
        self.cube_x = self.cube_coord.link_state.pose.position.x
        self.cube_y = self.cube_coord.link_state.pose.position.y
        self.cube_z = self.cube_coord.link_state.pose.position.z - 0.435

        # Goal and desired end-effector position
        self.x_goal = np.array([100*self.cube_x, 100*self.cube_y, 100*self.cube_z]) #only interested in position and not orientation
        self.x_d = np.array([self.x_goal[0],self.x_goal[1],self.x_goal[2],0,0,0]) #need orientation for proper dimensions

        self.obs = self.get_observation()
        self.reward = 0
        self.prev_reward = 0
        self.stage = 0
        self.pause()

        return self.obs

    def calculate_reward(self, new_obs):

        self.reward = 0
        self.new_obs = new_obs
        self.new_x0 = self.new_obs[0:3]
        self.x_goal = self.new_obs[-3:]
```

```

        self.diff_x = self.new_x0[0] - self.x_goal[0]
        self.diff_y = self.new_x0[1] - self.x_goal[1]
        self.diff_z = self.new_x0[2] - self.x_goal[2]

        self.distToGoal = np.linalg.norm(self.x_goal - self.new_x0)
        print("\nDist to goal = ", self.distToGoal)
        self.reward = -self.distToGoal

        if self.distToGoal <= 2.5:#3.5:
            self.reward += 2000#1000
            if np.linalg.norm(self.new_x0[0] - self.x_goal[0]) < 0.5:
                self.reward += 200
            if np.linalg.norm(self.new_x0[1] - self.x_goal[1]) < 0.5:
                self.reward += 200
            self.done = True
            self.done_counter +=1
            print("\ndone_counter =", self.done_counter)

        else:
            self.done = False

        print("\nReward: ", self.reward)

        self.info = np.array([self.diff_x, self.diff_y, self.diff_z])

        return self.reward, self.done, self.info

    def step(self,action):

        self.pause()
```

```
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point = JointTrajectoryPoint()

        self.action = action
        print("\nAction: ", self.action)

        # Impedance Stiffness and Damping
        self.Ki = np.diag(np.append(np.array(action), [1000, 1000, 1000]))
        self.Di = self.eps*self.Ki

        # Get measured joint position and velocity
        self.q_m = np.array(self.jointstate.position)
        self.q_m_r = self.q_m.reshape((-1,1))
        self.qdot_m = np.array(self.jointstate.velocity)
        self.qdot_m_r = self.qdot_m.reshape((-1,1))
        self.Te = np.array(self.robot.fkine(self.q_m))

        # Measured and Desired
        self.x_m = 100*np.array([self.Te[0][3],self.Te[1][3],self.Te[2][3],0,0,0]).reshape((-1,1))
        self.x_d = self.x_d.reshape((-1,1))

        self.J = self.robot.jacob0(self.q_m) # Jacobian matrix

        # Measured end-effector Velocity
        self.xdot_m = np.matmul(self.J,self.qdot_m_r)
```

```
        self.xdot_m = self.xdot_m

        # Actual and Desired Task Space Dynamics
        self.lambda_x = self.robot_dh.inertia_x(self.q_m) # Inertia Matrix
        self.mu_x = self.robot.coriolis_x(q = self.q_m[0:], qd = self.
    qdot_m[0:], Mx = self.lambda_x) #
    Coriolis
        self.gamma_x = self.robot.gravload_x(q = self.q_m).reshape
    ((-1,1)) #
    Gravity

        # Impedance Control
        self.mm1 = np.matmul(self.mu_x, self.xdot_m)
        self.xdm = self.x_d - self.x_m
        self.mm2 = np.matmul(self.Ki, self.xdm)
        self.mm3 = np.matmul(self.Di, self.xdot_m)
        self.W_e = self.mm1 + self.gamma_x + self.mm2 - self.mm3

        # Admittance control
        self.a = np.matmul(self.Ka, -self.xdm) + np.matmul(self.Da, self
    .xdot_m)
        self.b = self.W_e - self.a
        self.xddot_ac = np.matmul(np.linalg.inv(self.Md_a), self.b)

        # Acceleration to Position
        self.x_c = self.xdot_m*self.t + self.xddot_ac*(self.t**2)
        self.x_c = 0.01*np.reshape(self.x_c, 6)
        self.x_c = np.clip(self.x_c, np.array
    ([-0.5,-0.5,-0.5,-0.5,-0.5,-0.5]), np.array
    ([0.5,0.5,0.5,0.5,0.5,0.5]))

        self.x_c[0] += self.Te[0][3]
```

```
        self.x_c[1] += self.Te[1][3]
        self.x_c[2] += self.Te[2][3]

        self.x_cliped = np.clip(self.x_c, self.min_x, self.max_x)
        print("\nx_cliped: ", self.x_cliped)

        self.og_Te[0][3] = self.x_cliped[0]
        self.og_Te[1][3] = self.x_cliped[1]
        self.og_Te[2][3] = self.x_cliped[2]

        # Calculate joint positions
        self.sol = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q0)
        self.point.positions = self.sol.q

        # Publish UR5 velocity and position
        self.unpause()
        self.point.time_from_start = rospy.Duration(self.t)
        self.q_cmd.points.append(self.point)
        self.ur_cmd.publish(self.q_cmd)
        time.sleep(0.5)
        # time.sleep(1.5)
        self.pause()

        self.new_obs = self.get_observation()
        self.reward, self.done, self.info = self.calculate_reward(self.new_obs)

        # self.info = None

        return self.new_obs, self.reward, self.done, self.info
```

Listing 2: Variable Impedance Reaching Environment

Appendix C Variable Impedance Lifting Environment

```
#!/usr/bin/env python

# Gazebo Imports
import rospy
import rospkg
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState, GetModelState, GetLinkState,
    SetLinkProperties
import control_msgs.msg
import actionlib
from trajectory_msgs.msg import *
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint
from geometry_msgs.msg import WrenchStamped, Pose
from std_srvs.srv import Empty

import numpy as np
import gymnasium as gym
import sys
import torch
import time

# Robotics toolbox -python imports for kinematics and dynamics of ur5
import roboticstoolbox as rtb
from spatialmath import SE3

class UR_IMP_LIFT():

```

```python
    def __init__(self):

        rospy.init_node('ROBO_ENV', anonymous = True) # Initializing node

        self.jointstate = JointState()
        self.modelstate = ModelState()
        self.com = Pose()
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', '
    ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', '
    ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', '
    ur5_arm_wrist_3_joint']
        self.point = JointTrajectoryPoint()

        self.cube_name = 'cube1'
        self.cube_relative_entity_name = 'link'
        self.link_name = 'robot::left_inner_finger'
        self.robot = rtb.models.UR5() # Load UR5
        self.robot_dh = rtb.models.DH.UR5()

        # Gazebo Services
        self.model_coordinates = rospy.ServiceProxy('/gazebo/
    get_model_state', GetModelState)
        self.link_coordinates = rospy.ServiceProxy('/gazebo/
    get_link_state', GetLinkState)
        self.set_state = rospy.ServiceProxy('/gazebo/set_model_state',
    SetModelState)
        self.link_properties = rospy.ServiceProxy('/gazebo/
    set_link_properties', SetLinkProperties)
        self.unpause = rospy.ServiceProxy('/gazebo/unpause_physics',
    Empty)
```

```
        self.pause = rospy.ServiceProxy('/gazebo/pause_physics', Empty)

        # Publisher and Subscriber
        self.ur_cmd = rospy.Publisher('/arm_controller/command',
    JointTrajectory, queue_size = 1)
        self.ur_jointstate = rospy.Subscriber('/joint_states',
    JointState, self.ur5_joint_callback)
        self.gripper_client = actionlib.SimpleActionClient('/
    gripper_controller/gripper_cmd', control_msgs.msg.
    GripperCommandAction) #.0/.8:open/close
        self.ft_sensor = rospy.Subscriber('/ft_sensor/raw',
    WrenchStamped, self.ft_sensor_callback)
        self.goal = control_msgs.msg.GripperCommandGoal()

        # Limits of end-effector position
        self.max = np.array([0.60, 0.22, 0.50, 0, 0, 0])#30])
        self.min = np.array([0.29, -0.22, 0.22, 0, 0, 0])#188])
        self.max_x = torch.tensor(self.max, dtype = torch.float32,
    device = torch.device("cpu"))
        self.min_x = torch.tensor(self.min, dtype = torch.float32,
    device = torch.device("cpu"))

        # Action space:  x direction,y direction,z direction:  task space
        self.action_space = gym.spaces.Box(low = np.array([-12,-12,-12])
    , high = np.array([12,12,12]), dtype= np.float32)
        self.max_action = self.action_space.high
        self.min_action = self.action_space.low

        # Observation Space = [x,y,z,goal.x,goal.y,goal.z]
        self.observation_space = gym.spaces.Box(low = np.array([29, -22,
     22, 29, -22, 70]), high = np.array([70, 22, 95, 70, 22, 95]), dtype
```

```
=np.float32)

        #self.cuda0 = torch.device('cuda:0')

        self.reward = 0
        self.prev_reward = 0
        self.prev_distToGoal = 0
        self.distToGoal = 0
        self.done_counter = 0
        self.eps = 10#0.75
        self.Ka = 1*np.identity(6)
        self.Da = self.eps*self.Ka
        self.Md_a = 3*np.identity(6)
        self.t = 0.2
        self.gravity_acc = np.array([0,0,9.81,0,0,0]).reshape((-1,1))

        # Desired Velocity and Acceleration
        self.xdot_d = np.zeros(6,).reshape((-1,1))
        self.xddot_d = np.zeros(6,).reshape((-1,1))



    def ur5_joint_callback(self, data):

        self.jointstate = data

    def ft_sensor_callback(self,data):

        self.ft_data = data

    def get_observation(self):

```

```
        self.q0 = self.jointstate.position

        # Cube Coordinates
        self.inner_finger_coord = self.link_coordinates(self.link_name, 'world')
        self.tcp_x = self.inner_finger_coord.link_state.pose.position.x - 0.0681975
        self.tcp_y = self.inner_finger_coord.link_state.pose.position.y
        self.tcp_z = self.inner_finger_coord.link_state.pose.position.z
        self.tcp_coord = np.array([100*self.tcp_x, 100*self.tcp_y, 100*self.tcp_z])
        print("\nTCP Coordinates: ", self.tcp_coord)

        # Creating observation array
        self.obs = np.array([])
        self.obs = np.append(self.obs, self.tcp_coord)
        self.obs = np.append(self.obs, self.x_goal)

        return self.obs


    def reset(self):

        self.q_cmd1 = JointTrajectory()
        self.q_cmd1.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point1 = JointTrajectoryPoint()
        self.point2 = JointTrajectoryPoint()
        self.q = [0.0,-1.57,1.57,-1.57,-1.57,1.57]
```

```

        self.goal_x = 45
        self.goal_y = 11 #np.random.uniform(-22, 22)
        self.goal_z = 88 #np.random.uniform(45, 89)
        self.x_goal = np.array([self.goal_x, self.goal_y, self.goal_z])
        self.x_d = np.array([self.goal_x, self.goal_y, self.goal_z, 0, 0, 0])

        # Release the cube
        self.unpause()
        self.gripper_status = 0
        self.gripper_client.wait_for_server()
        self.goal.command.position = self.gripper_status
        self.goal.command.max_effort = -1.0 # Do not limit the effort
        self.gripper_client.send_goal(self.goal)
        self.gripper_client.wait_for_result()
        time.sleep(1.0)
        self.pause()

        # Randomize mass of cube and set link properties
        self.mass = np.random.uniform(1,4)
        print("\nmass: ",self.mass)
        self.inertia = (1/12)*self.mass*(0.05**2+0.5**2)
        self.gravity_mode = True
        self.com.position.x = 0.0
        self.com.position.y = 0.0
        self.com.position.z = 0.0
        self.com.orientation.x = 0.0
        self.com.orientation.y = 0.0
        self.com.orientation.z = 0.0
        self.com.orientation.w = 0.0
```

```python
        self.ixx = self.inertia
        self.ixy = 0
        self.ixz = 0
        self.iyy = self.inertia
        self.iyz = 0
        self.izz = self.inertia

        rospy.wait_for_service('/gazebo/set_link_properties')

        try:
            # self.unpause()
            self.resp1 = self.link_properties('cube1::link', self.com,
    self.gravity_mode, self.mass, self.ixx, self.ixy, self.ixz, self.iyy
    , self.iyz, self.izz)
            time.sleep(0.3)
            # self.pause()

        except rospy.ServiceException as e:
            print ("Service call failed: %s" % e)

        # Randomize x and y location of cube
        self.cube_x = np.random.uniform(0.3,0.6)
        self.cube_y = np.random.uniform(-0.22,0.22)
        self.modelstate.model_name = 'cube1'
        self.modelstate.pose.position.x = self.cube_x
        self.modelstate.pose.position.y = self.cube_y
        self.modelstate.pose.position.z = 0.6
        self.modelstate.pose.orientation.x = 0
        self.modelstate.pose.orientation.y = 0
        self.modelstate.pose.orientation.z = 0
        self.modelstate.pose.orientation.w = 0
```

```
        self.unpause()
        rospy.wait_for_service('/gazebo/set_model_state')

        try:
            self.resp = self.set_state(self.modelstate)
            # time.sleep(0.3)
            time.sleep(0.6)

        except rospy.ServiceException as e:
            print ("Service call failed: %s" % e)

        self.pause()

        # UR5 reset position
        self.q_dot_cmd = [0.0,0.0,0.0,0.0,0.0,0.0]
        self.og_Te = np.array(self.robot.fkine(np.array
    ([0.0,-1.57,1.57,-1.57,-1.57,1.57])))
        self.sol1 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q)
        self.point1.positions = self.sol1.q
        self.point1.velocities = -3*self.q_dot_cmd
        self.point1.time_from_start = rospy.Duration(1)
        self.q_cmd1.points.append(self.point1)

        # Move UR5 gripper to where the cube is
        self.ur_x = self.cube_x
        self.ur_y = self.cube_y
        self.og_Te[0][3] = self.ur_x
        self.og_Te[1][3] = self.ur_y
        self.og_Te[2][3] = 0.215
        self.sol2 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.sol1.
    q)
```

```
        self.point2.positions = self.sol2.q

        # Publish UR5 velocity and position
        self.point2.velocities = -3*self.q_dot_cmd
        self.point2.time_from_start = rospy.Duration(2)
        self.q_cmd1.points.append(self.point2)
        self.unpause()
        self.ur_cmd.publish(self.q_cmd1)
        time.sleep(1)
        # time.sleep(2)

        # Grasp the object
        self.gripper_status = 0.8
        self.gripper_client.wait_for_server()
        self.goal.command.position = self.gripper_status
        self.goal.command.max_effort = -1.0 # Do not limit the effort
        self.gripper_client.send_goal(self.goal)
        time.sleep(1.5)

        self.obs = self.get_observation()
        self.reward = 0
        self.prev_reward = 0
        self.stage = 0
        self.pause()

        return self.obs

    def calculate_reward(self, new_obs):

        self.reward = 0
        self.new_obs = new_obs
```

```python
        self.new_x0 = self.new_obs[0:3]
        self.x_goal = self.new_obs[-3:]
        print("\nx_goal: ", self.x_goal)

        self.diff_x = self.new_x0[0] - self.x_goal[0]
        self.diff_y = self.new_x0[1] - self.x_goal[1]
        self.diff_z = self.new_x0[2] - self.x_goal[2]

        self.distToGoal = np.linalg.norm(self.x_goal - self.new_x0)
        print("\nDist to goal = ", self.distToGoal)
        self.reward = -self.distToGoal

        if self.distToGoal <= 3.5:
            self.reward += 2000#1000
            if np.linalg.norm(self.new_x0[0] - self.x_goal[0]) < 1:
                self.reward += 200
            if np.linalg.norm(self.new_x0[1] - self.x_goal[1]) < 1:
                self.reward += 200
            self.done = True
            self.done_counter +=1
            print("\ndone_counter =", self.done_counter)

        else:
            self.done = False

        print("\nReward: ", self.reward)

        self.info = np.array([self.diff_x, self.diff_y, self.diff_z])

        return self.reward, self.done, self.info

```

```
    def step(self,action):

        self.pause()
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point = JointTrajectoryPoint()

        self.action = action
        print("\nAction: ", self.action)

        # Impedance Stiffness and Damping
        self.Ki = np.diag(np.append(np.array(action), [1000, 1000, 1000]))
        self.Di = self.eps*self.Ki

        # Get measured joint position and velocity
        self.q_m = np.array(self.jointstate.position)
        self.q_m_r = self.q_m.reshape((-1,1))
        self.qdot_m = np.array(self.jointstate.velocity)
        self.qdot_m_r = self.qdot_m.reshape((-1,1))
        self.Te = np.array(self.robot.fkine(self.q_m))

        # Measured and Desired
        self.x_m = 100*np.array([self.Te[0][3],self.Te[1][3],self.Te[2][3]+0.445,0,0,0]).reshape((-1,1))
        self.x_d = self.x_d.reshape((-1,1))


```

```python
        self.J = self.robot.jacob0(self.q_m) # Jacobian matrix

        # Measured end-effector Velocity
        self.xdot_m = np.matmul(self.J,self.qdot_m_r)
        self.xdot_m = self.xdot_m


        # Actual and Desired Task Space Dynamics
        self.lambda_x = self.robot_dh.inertia_x(self.q_m) # Inertia Matrix

        self.mu_x = self.robot.coriolis_x(q = self.q_m[0:], qd = self.qdot_m[0:], Mx = self.lambda_x) # Coriolis

        self.gamma_x = self.robot.gravload_x(q = self.q_m).reshape((-1,1)) # Gravity


        # Impedance Control
        self.mm1 = np.matmul(self.mu_x, self.xdot_m)

        self.xdm = self.x_d - self.x_m
        self.mm2 = np.matmul(self.Ki, self.xdm)
        self.mm3 = np.matmul(self.Di, self.xdot_m)
        self.W_e = self.mm1 + self.gamma_x + self.mm2 - self.mm3

        # Admittance control
        self.a = np.matmul(self.Ka, self.xdm) + np.matmul(self.Da, self.xdot_m)
        self.mm4 = self.mass*self.gravity_acc
        self.b = self.W_e - self.mm4 - self.a
```

```
        self.xddot_ac = np.matmul(np.linalg.inv(self.Md_a), self.b)

        # Acceleration to Position
        self.x_c = self.xdot_m*self.t + self.xddot_ac*(self.t**2)
        self.x_c = 0.01*np.reshape(self.x_c, 6)
        self.x_c = np.clip(self.x_c, np.array
    ([-0.2,-0.2,-0.2,-0.2,-0.2,-0.2]), np.array
    ([0.2,0.2,0.2,0.2,0.2,0.2]))


        self.x_c[0] += self.Te[0][3]
        self.x_c[1] += self.Te[1][3]
        self.x_c[2] += self.Te[2][3]

        self.x_cliped = np.clip(self.x_c, self.min_x, self.max_x)
        print("\nx_cliped: ", self.x_cliped)

        self.og_Te[0][3] = self.x_cliped[0]
        self.og_Te[1][3] = self.x_cliped[1]
        self.og_Te[2][3] = self.x_cliped[2]

        # Calculate joint positions
        self.sol = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q0)
        self.point.positions = self.sol.q

        # Publish UR5 velocity and position
        self.unpause()
        self.point.time_from_start = rospy.Duration(self.t)
        self.q_cmd.points.append(self.point)
        self.ur_cmd.publish(self.q_cmd)
        time.sleep(0.5)
```

```
        # time.sleep(1.5)
        self.pause()

        self.new_obs = self.get_observation()
        self.reward, self.done, self.info = self.calculate_reward(self.
    new_obs)

        # self.info = None

        return self.new_obs, self.reward, self.done, self.info
```

Listing 3: Variable Impedance Lifting Environment

Appendix D Variable PD Reaching Environment

```
#!/usr/bin/env python

# Gazebo Imports
import rospy
import rospkg
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState, GetModelState, GetLinkState
import control_msgs.msg
import actionlib
from trajectory_msgs.msg import *
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint
from geometry_msgs.msg import WrenchStamped
from std_srvs.srv import Empty

import numpy as np
import gymnasium as gym
import sys
import torch
import time

# Robotics toolbox -python imports for kinematics and dynamics of ur5
import roboticstoolbox as rtb
from spatialmath import SE3

class UR_PD_REACH():

    def __init__(self):
```

```python

        rospy.init_node('ROBO_ENV', anonymous = True) # Initializing node

        self.jointstate = JointState()
        self.modelstate = ModelState()
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point = JointTrajectoryPoint()

        self.cube_name = 'cube1'
        self.cube_relative_entity_name = 'link'
        self.link_name = 'robot::left_inner_finger'
        self.robot = rtb.models.UR5() # Load UR5
        self.robot_dh = rtb.models.DH.UR5()

        # Gazebo Services
        self.model_coordinates = rospy.ServiceProxy('/gazebo/get_model_state', GetModelState)
        self.link_coordinates = rospy.ServiceProxy('/gazebo/get_link_state', GetLinkState)
        self.set_state = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)
        self.unpause = rospy.ServiceProxy('/gazebo/unpause_physics', Empty)
        self.pause = rospy.ServiceProxy('/gazebo/pause_physics', Empty)

        # Publisher and Subscriber
```

```
        self.ur_cmd = rospy.Publisher('/arm_controller/command', JointTrajectory, queue_size = 1)
        self.ur_jointstate = rospy.Subscriber('/joint_states', JointState, self.ur5_joint_callback)
        self.gripper_client = actionlib.SimpleActionClient('/gripper_controller/gripper_cmd', control_msgs.msg.GripperCommandAction)
        self.ft_sensor = rospy.Subscriber('/ft_sensor/raw', WrenchStamped, self.ft_sensor_callback)
        self.goal = control_msgs.msg.GripperCommandGoal()

        # Limits of end-effector position
        self.max = np.array([0.60, 0.22, 0.40, 0, 0, 0])
        self.min = np.array([0.29, -0.22, 0.2, 0, 0, 0])
        self.max_x = torch.tensor(self.max, dtype = torch.float32, device = torch.device("cpu"))
        self.min_x = torch.tensor(self.min, dtype = torch.float32, device = torch.device("cpu"))

        self.action_space = gym.spaces.Box(low = np.array([-6,-6,-6]), high = np.array([6,6,6]), dtype= np.float32)
        self.max_action = self.action_space.high
        self.min_action = self.action_space.low

        # Observation Space = [x,y,z,cube.x,cube.y,cube.z]
        self.observation_space = gym.spaces.Box(low = np.array([30, -25, 20, 40, -15, 0]), high = np.array([70, 25, 35, 50, 15, 60]), dtype= np.float32)

        self.cuda0 = torch.device('cuda:0')

```

```
        self.reward = 0
        self.prev_reward = 0
        self.prev_distToGoal = 0
        self.distToGoal = 0
        self.done_counter = 0
        self.eps = 0.75
        self.Ka = 1*np.identity(6)
        self.Da = self.eps*self.Ka
        self.Md_a = 3*np.identity(6)
        self.t = 0.5

        # Desired Velocity and Acceleration
        self.xdot_d = np.zeros(6,).reshape((-1,1))
        self.xddot_d = np.zeros(6,).reshape((-1,1))

    def ur5_joint_callback(self, data):

        self.jointstate = data

    def ft_sensor_callback(self,data):

        self.ft_data = data

    def get_observation(self):

        self.q0 = self.jointstate.position

        # Cube Coordinates
        self.inner_finger_coord = self.link_coordinates(self.link_name,
    'world')
```

```
        self.tcp_x = self.inner_finger_coord.link_state.pose.position.x - 0.0681975
        self.tcp_y = self.inner_finger_coord.link_state.pose.position.y
        self.tcp_z = self.inner_finger_coord.link_state.pose.position.z - 0.066 - 0.435
        self.tcp_coord = np.array([100*self.tcp_x, 100*self.tcp_y, 100*self.tcp_z])
        print("\nTCP Coordinates: ", self.tcp_coord)

        # Creating observation array
        self.obs = np.array([])
        self.obs = np.append(self.obs, self.tcp_coord)
        self.obs = np.append(self.obs, self.x_goal)

        return self.obs


    def reset(self):

        self.q_cmd1 = JointTrajectory()
        self.q_cmd2 = JointTrajectory()
        self.q_cmd1.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.q_cmd2.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point1 = JointTrajectoryPoint()
        self.point2 = JointTrajectoryPoint()
```

```

        self.q = [0.0,-1.57,1.57,-1.57,-1.57,1.57]

        # UR5 reset position
        self.q_dot_cmd = [0.0,0.0,0.0,0.0,0.0,0.0]
        self.og_Te = np.array(self.robot.fkine(np.array([0.0,-1.57,1.57,-1.57,-1.57,1.57])))
        self.sol1 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q)
        self.point1.positions = self.sol1.q
        self.point1.velocities = -3*self.q_dot_cmd
        self.point1.time_from_start = rospy.Duration(1)
        self.q_cmd1.points.append(self.point1)

        # Randomize UR5 gripper x and y location
        self.ur_x = np.random.uniform(0.30,0.59)
        self.ur_y = np.random.uniform(-0.15,0.15)
        self.og_Te[0][3] = self.ur_x
        self.og_Te[1][3] = self.ur_y
        self.sol2 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.sol1.q)
        self.point2.positions = self.sol2.q

        # Publish UR5 velocity and position
        self.unpause()
        self.point2.velocities = -3*self.q_dot_cmd
        self.point2.time_from_start = rospy.Duration(2)
        self.q_cmd1.points.append(self.point2)
        self.ur_cmd.publish(self.q_cmd1)
        time.sleep(0.5)
        # time.sleep(3)

```

```
        # Publish gripper as open and set gripper status :  0
        self.gripper_status = 0
        self.gripper_client.wait_for_server()
        self.goal.command.position = self.gripper_status
        self.goal.command.max_effort = -1.0 # Do not limit the effort
        self.gripper_client.send_goal(self.goal)
        self.gripper_client.wait_for_result()

        # Randomize x and y location of cube
        self.cube_x = np.random.uniform(0.4,0.6)
        self.cube_y = np.random.uniform(-0.15,0.15)

        # Cube reset position
        self.modelstate.model_name = 'cube1'
        self.modelstate.pose.position.x = self.cube_x #0.4
        self.modelstate.pose.position.y = self.cube_y #-0.1
        self.modelstate.pose.position.z = 0.6
        self.modelstate.pose.orientation.x = 0
        self.modelstate.pose.orientation.y = 0
        self.modelstate.pose.orientation.z = 0
        self.modelstate.pose.orientation.w = 0
        rospy.wait_for_service('/gazebo/set_model_state')

        try:
            self.resp = self.set_state(self.modelstate)
            # time.sleep(0.3)
            time.sleep(0.5)

        except rospy.ServiceException as e:
            print ("Service call failed: %s" % e)

```

```python
        self.cube_coord = self.link_coordinates('cube1::link', 'world')
        self.cube_x = self.cube_coord.link_state.pose.position.x
        self.cube_y = self.cube_coord.link_state.pose.position.y
        self.cube_z = self.cube_coord.link_state.pose.position.z - 0.435

        # Goal and desired end-effector position
        self.x_goal = np.array([100*self.cube_x, 100*self.cube_y, 100*self.cube_z]) # only interested in position and not orientation
        self.x_d = np.array([self.x_goal[0],self.x_goal[1],self.x_goal[2],0,0,0]) # need orientation for proper dimensions

        self.obs = self.get_observation()
        self.reward = 0
        self.prev_reward = 0
        self.stage = 0
        self.pause()

        return self.obs

    def calculate_reward(self, new_obs):

        self.reward = 0
        self.new_obs = new_obs
        self.new_x0 = self.new_obs[0:3]
        self.x_goal = self.new_obs[-3:]

        self.diff_x = self.new_x0[0] - self.x_goal[0]
        self.diff_y = self.new_x0[1] - self.x_goal[1]
        self.diff_z = self.new_x0[2] - self.x_goal[2]

```

```
        self.distToGoal = np.linalg.norm(self.x_goal - self.new_x0)
        print("\nDist to goal = ", self.distToGoal)
        self.reward = -self.distToGoal

        if self.distToGoal <= 2.5:#3.5:
            self.reward += 2000#1000
            if np.linalg.norm(self.new_x0[0] - self.x_goal[0]) < 0.5:
                self.reward += 200
            if np.linalg.norm(self.new_x0[1] - self.x_goal[1]) < 0.5:
                self.reward += 200
            self.done = True
            self.done_counter +=1
            print("\ndone_counter =", self.done_counter)

        else:
            self.done = False

        print("\nReward: ", self.reward)

        self.info = np.array([self.diff_x, self.diff_y, self.diff_z])

        return self.reward, self.done, self.info

    def step(self,action):

        self.pause()
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', '
    ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', '
    ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', '
    ur5_arm_wrist_3_joint']
```

```
        self.point = JointTrajectoryPoint()

        self.action = action
        print("\nAction: ", self.action)

        # P: Ki and D: Di
        self.Ki = np.diag(np.append(np.array(action), [1000, 1000, 1000]))
        self.Di = self.eps*self.Ki

        # Get measured joint position and velocity
        self.q_m = np.array(self.jointstate.position)
        self.q_m_r = self.q_m.reshape((-1,1))
        self.qdot_m = np.array(self.jointstate.velocity)
        self.qdot_m_r = self.qdot_m.reshape((-1,1))
        self.Te = np.array(self.robot.fkine(self.q_m))

        # Measured and Desired
        self.x_m = 100*np.array([self.Te[0][3],self.Te[1][3],self.Te[2][3],0,0,0]).reshape((-1,1))
        self.x_d = self.x_d.reshape((-1,1))

        self.J = self.robot.jacob0(self.q_m) # Jacobian matrix

        # Measured end-effector Velocity
        self.xdot_m = np.matmul(self.J,self.qdot_m_r)
        self.xdot_m = self.xdot_m

        # Impedance Control
        self.xdm = self.x_d - self.x_m
```

```
        self.W_e = np.matmul(self.Ki, self.xdm) - np.matmul(self.Di, self.xdot_m)

        # Admittance control
        self.a = np.matmul(self.Ka, -self.xdm) + np.matmul(self.Da, self.xdot_m)
        self.b = self.W_e - self.a
        self.xddot_ac = np.matmul(np.linalg.inv(self.Md_a), self.b)

        # Acceleration to Position
        self.x_c = self.xdot_m*self.t + self.xddot_ac*(self.t**2)
        self.x_c = 0.01*np.reshape(self.x_c, 6)
        self.x_c = np.clip(self.x_c, np.array([-0.5,-0.5,-0.5,-0.5,-0.5,-0.5]), np.array([0.5,0.5,0.5,0.5,0.5,0.5]))

        self.x_c[0] += self.Te[0][3]
        self.x_c[1] += self.Te[1][3]
        self.x_c[2] += self.Te[2][3]

        self.x_cliped = np.clip(self.x_c, self.min_x, self.max_x)
        print("\nx_cliped: ", self.x_cliped)

        self.og_Te[0][3] = self.x_cliped[0]
        self.og_Te[1][3] = self.x_cliped[1]
        self.og_Te[2][3] = self.x_cliped[2]

        # Calculate joint positions
        self.sol = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q0)
        self.point.positions = self.sol.q

```

```
        # Publish UR5 velocity and position
        self.unpause()
        self.point.time_from_start = rospy.Duration(self.t)
        self.q_cmd.points.append(self.point)
        self.ur_cmd.publish(self.q_cmd)
        time.sleep(0.5)
        # time.sleep(1.5)
        self.pause()

        self.new_obs = self.get_observation()
        self.reward, self.done, self.info = self.calculate_reward(self.new_obs)

        # self.info = None

        return self.new_obs, self.reward, self.done, self.info
```

Listing 4: Variable PD Reaching Environment

Appendix E Variable PD Lifting Environment

```
#!/usr/bin/env python

# Gazebo Imports
import rospy
import rospkg
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState, GetModelState, GetLinkState,
    SetLinkProperties
import control_msgs.msg
import actionlib
from trajectory_msgs.msg import *
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint
from geometry_msgs.msg import WrenchStamped, Pose
from std_srvs.srv import Empty

import numpy as np
import gymnasium as gym
import sys
import torch
import time

# Robotics toolbox -python imports for kinematics and dynamics of ur5
import roboticstoolbox as rtb
from spatialmath import SE3

class UR_PD_LIFT():

```

```
    def __init__(self):

        rospy.init_node('ROBO_ENV', anonymous = True) # Initializing node

        self.jointstate = JointState()
        self.modelstate = ModelState()
        self.com = Pose()
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point = JointTrajectoryPoint()

        self.cube_name = 'cube1'
        self.cube_relative_entity_name = 'link'
        self.link_name = 'robot::left_inner_finger'
        self.robot = rtb.models.UR5() # Load UR5
        self.robot_dh = rtb.models.DH.UR5()

        # Gazebo Services
        self.model_coordinates = rospy.ServiceProxy('/gazebo/get_model_state', GetModelState)
        self.link_coordinates = rospy.ServiceProxy('/gazebo/get_link_state', GetLinkState)
        self.set_state = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)
        self.link_properties = rospy.ServiceProxy('/gazebo/set_link_properties', SetLinkProperties)
        self.unpause = rospy.ServiceProxy('/gazebo/unpause_physics', Empty)
```

```
        self.pause = rospy.ServiceProxy('/gazebo/pause_physics', Empty)

        # Publisher and Subscriber
        self.ur_cmd = rospy.Publisher('/arm_controller/command',
    JointTrajectory, queue_size = 1)
        self.ur_jointstate = rospy.Subscriber('/joint_states',
    JointState, self.ur5_joint_callback)
        self.gripper_client = actionlib.SimpleActionClient('/
    gripper_controller/gripper_cmd', control_msgs.msg.
    GripperCommandAction)
        self.ft_sensor = rospy.Subscriber('/ft_sensor/raw',
    WrenchStamped, self.ft_sensor_callback)
        self.goal = control_msgs.msg.GripperCommandGoal()

        # Limits of end-effector position
        self.max = np.array([0.60, 0.22, 0.50, 0, 0, 0])#30])
        self.min = np.array([0.29, -0.22, 0.22, 0, 0, 0])#188])
        self.max_x = torch.tensor(self.max, dtype = torch.float32,
    device = torch.device("cpu"))
        self.min_x = torch.tensor(self.min, dtype = torch.float32,
    device = torch.device("cpu"))

        # Action space = [x,y,z]
        self.action_space = gym.spaces.Box(low = np.array([-60,-60,-60])
    , high = np.array([60,60,60]), dtype= np.float32)
        self.max_action = self.action_space.high
        self.min_action = self.action_space.low

        # Observation Space = [x,y,z,goal.x,goal.y,goal.z]
        self.observation_space = gym.spaces.Box(low = np.array([29, -22,
     22, 29, -22, 70]), high = np.array([70, 22, 95, 70, 22, 95]), dtype
```

```
=np.float32)

        #self.cuda0 = torch.device('cuda:0')

        self.reward = 0
        self.prev_reward = 0
        self.prev_distToGoal = 0
        self.distToGoal = 0
        self.done_counter = 0
        self.eps = 10#0.75
        self.Ka = 1*np.identity(6)
        self.Da = self.eps*self.Ka
        self.Md_a = 3*np.identity(6)
        self.t = 0.2
        self.gravity_acc = np.array([0,0,9.81,0,0,0]).reshape((-1,1))

        # Desired Velocity and Acceleration
        self.xdot_d = np.zeros(6,).reshape((-1,1))
        self.xddot_d = np.zeros(6,).reshape((-1,1))


    def ur5_joint_callback(self, data):

        self.jointstate = data

    def ft_sensor_callback(self,data):

        self.ft_data = data

    def get_observation(self):

```

```
        self.q0 = self.jointstate.position

        # Cube Coordinates
        self.inner_finger_coord = self.link_coordinates(self.link_name, 'world')
        self.tcp_x = self.inner_finger_coord.link_state.pose.position.x - 0.0681975
        self.tcp_y = self.inner_finger_coord.link_state.pose.position.y
        self.tcp_z = self.inner_finger_coord.link_state.pose.position.z
        self.tcp_coord = np.array([100*self.tcp_x, 100*self.tcp_y, 100*self.tcp_z])
        print("\nTCP Coordinates: ", self.tcp_coord)

        # Creating observation array
        self.obs = np.array([])
        self.obs = np.append(self.obs, self.tcp_coord)
        self.obs = np.append(self.obs, self.x_goal)

        return self.obs


    def reset(self):

        self.q_cmd1 = JointTrajectory()
        self.q_cmd1.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point1 = JointTrajectoryPoint()
        self.point2 = JointTrajectoryPoint()
        self.q = [0.0,-1.57,1.57,-1.57,-1.57,1.57]
```

```
        self.goal_x = 45
        self.goal_y = 11 #np.random.uniform(-22, 22)
        self.goal_z = 88 #np.random.uniform(45, 89)
        self.x_goal = np.array([self.goal_x, self.goal_y, self.goal_z])
        self.x_d = np.array([self.goal_x, self.goal_y, self.goal_z, 0,
    0, 0])

        # Release the cube
        self.unpause()
        self.gripper_status = 0
        self.gripper_client.wait_for_server()
        self.goal.command.position = self.gripper_status
        self.goal.command.max_effort = -1.0 # Do not limit the effort
        self.gripper_client.send_goal(self.goal)
        self.gripper_client.wait_for_result()
        time.sleep(1.0)
        self.pause()

        # Randomize mass of cube and set link properties
        self.mass = np.random.uniform(1,2.5)
        print("\nmass: ",self.mass)
        self.inertia = (1/12)*self.mass*(0.05**2+0.5**2)
        self.gravity_mode = True
        self.com.position.x = 0.0
        self.com.position.y = 0.0
        self.com.position.z = 0.0
        self.com.orientation.x = 0.0
        self.com.orientation.y = 0.0
        self.com.orientation.z = 0.0
        self.com.orientation.w = 0.0
```

```
        self.ixx = self.inertia
        self.ixy = 0
        self.ixz = 0
        self.iyy = self.inertia
        self.iyz = 0
        self.izz = self.inertia

        rospy.wait_for_service('/gazebo/set_link_properties')

        try:
            # self.unpause()
            self.resp1 = self.link_properties('cube1::link', self.com,
    self.gravity_mode, self.mass, self.ixx, self.ixy, self.ixz, self.iyy
    , self.iyz, self.izz)
            time.sleep(0.3)
            # self.pause()

        except rospy.ServiceException as e:
            print ("Service call failed: %s" % e)

        # Randomize x and y location of cube
        self.cube_x = np.random.uniform(0.3,0.6)
        self.cube_y = np.random.uniform(-0.22,0.22)
        self.modelstate.model_name = 'cube1'
        self.modelstate.pose.position.x = self.cube_x
        self.modelstate.pose.position.y = self.cube_y
        self.modelstate.pose.position.z = 0.6
        self.modelstate.pose.orientation.x = 0
        self.modelstate.pose.orientation.y = 0
        self.modelstate.pose.orientation.z = 0
        self.modelstate.pose.orientation.w = 0
```

```
        self.unpause()
        rospy.wait_for_service('/gazebo/set_model_state')

        try:
            self.resp = self.set_state(self.modelstate)
            # time.sleep(0.3)
            time.sleep(0.6)

        except rospy.ServiceException as e:
            print ("Service call failed: %s" % e)

        self.pause()

        # UR5 reset position
        self.q_dot_cmd = [0.0,0.0,0.0,0.0,0.0,0.0]
        self.og_Te = np.array(self.robot.fkine(np.array
    ([0.0,-1.57,1.57,-1.57,-1.57,1.57])))
        self.sol1 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q)
        self.point1.positions = self.sol1.q
        self.point1.velocities = -3*self.q_dot_cmd
        self.point1.time_from_start = rospy.Duration(1)
        self.q_cmd1.points.append(self.point1)

        # Move UR5 gripper to where the cube is
        self.ur_x = self.cube_x
        self.ur_y = self.cube_y
        self.og_Te[0][3] = self.ur_x
        self.og_Te[1][3] = self.ur_y
        self.og_Te[2][3] = 0.215
        self.sol2 = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.sol1.
    q)
```

```
        self.point2.positions = self.sol2.q

        # Publish UR5 velocity and position
        self.point2.velocities = -3*self.q_dot_cmd
        self.point2.time_from_start = rospy.Duration(2)
        self.q_cmd1.points.append(self.point2)
        self.unpause()
        self.ur_cmd.publish(self.q_cmd1)
        time.sleep(1)
        # time.sleep(2)

        # Grasp the object
        self.gripper_status = 0.8
        self.gripper_client.wait_for_server()
        self.goal.command.position = self.gripper_status
        self.goal.command.max_effort = -1.0 # Do not limit the effort
        self.gripper_client.send_goal(self.goal)
        time.sleep(1.5)

        self.obs = self.get_observation()
        self.reward = 0
        self.prev_reward = 0
        self.stage = 0
        self.pause()

        return self.obs

    def calculate_reward(self, new_obs):

        self.reward = 0
        self.new_obs = new_obs
```

```
        self.new_x0 = self.new_obs[0:3]
        self.x_goal = self.new_obs[-3:]
        print("\nx_goal: ", self.x_goal)

        self.diff_x = self.new_x0[0] - self.x_goal[0]
        self.diff_y = self.new_x0[1] - self.x_goal[1]
        self.diff_z = self.new_x0[2] - self.x_goal[2]

        self.distToGoal = np.linalg.norm(self.x_goal - self.new_x0)
        print("\nDist to goal = ", self.distToGoal)
        self.reward = -self.distToGoal

        if self.distToGoal <= 3.5:
            self.reward += 2000#1000
            if np.linalg.norm(self.new_x0[0] - self.x_goal[0]) < 1:
                self.reward += 200
            if np.linalg.norm(self.new_x0[1] - self.x_goal[1]) < 1:
                self.reward += 200
            self.done = True
            self.done_counter +=1
            print("\ndone_counter =", self.done_counter)

        else:
            self.done = False

        print("\nReward: ", self.reward)

        self.info = np.array([self.diff_x, self.diff_y, self.diff_z])

        return self.reward, self.done, self.info

```

```
    def step(self,action):

        self.pause()
        self.q_cmd = JointTrajectory()
        self.q_cmd.joint_names = ['ur5_arm_shoulder_pan_joint', 'ur5_arm_shoulder_lift_joint', 'ur5_arm_elbow_joint', 'ur5_arm_wrist_1_joint', 'ur5_arm_wrist_2_joint', 'ur5_arm_wrist_3_joint']
        self.point = JointTrajectoryPoint()

        self.action = action
        print("\nAction: ", self.action)

        # Impedance Stiffness and Damping
        self.Ki = np.diag(np.append(np.array(action), [1000, 1000, 1000]))
        self.Di = self.eps*self.Ki

        # Get measured joint position and velocity
        self.q_m = np.array(self.jointstate.position)
        self.q_m_r = self.q_m.reshape((-1,1))
        self.qdot_m = np.array(self.jointstate.velocity)
        self.qdot_m_r = self.qdot_m.reshape((-1,1))
        self.Te = np.array(self.robot.fkine(self.q_m))

        # Measured and Desired
        self.x_m = 100*np.array([self.Te[0][3],self.Te[1][3],self.Te[2][3]+0.445,0,0,0]).reshape((-1,1))
        self.x_d = self.x_d.reshape((-1,1))

        self.J = self.robot.jacob0(self.q_m) # Jacobian matrix
```

```python

        # Measured end-effector Velocity
        self.xdot_m = np.matmul(self.J,self.qdot_m_r)
        self.xdot_m = self.xdot_m

        # Actual and Desired Task Space Dynamics
        self.lambda_x = self.robot_dh.inertia_x(self.q_m) # Inertia Matrix
        self.mu_x = self.robot.coriolis_x(q = self.q_m[0:], qd = self.qdot_m[0:], Mx = self.lambda_x) # Coriolis
        self.gamma_x = self.robot.gravload_x(q = self.q_m).reshape((-1,1)) # Gravity

        # Impedance Control
        self.xdm = self.x_d - self.x_m
        print("\nxdm: ", self.xdm)
        self.W_e = np.matmul(self.Ki, self.xdm) - np.matmul(self.Di, self.xdot_m)
        print("\nW_e: ",self.W_e)

        # Admittance control
        self.a = np.matmul(self.Ka, self.xdm) + np.matmul(self.Da, self.xdot_m)
        self.mm4 = self.mass*self.gravity_acc
        self.b = self.W_e - self.mm4 - self.a
        self.xddot_ac = np.matmul(np.linalg.inv(self.Md_a), self.b)


        # Acceleration to Position
        self.x_c = self.xdot_m*self.t + self.xddot_ac*(self.t**2)
        self.x_c = 0.01*np.reshape(self.x_c, 6)
```

```
        self.x_c = np.clip(self.x_c, np.array
    ([-0.2,-0.2,-0.2,-0.2,-0.2,-0.2]), np.array
    ([0.2,0.2,0.2,0.2,0.2,0.2]))

        self.x_c[0] += self.Te[0][3]
        self.x_c[1] += self.Te[1][3]
        self.x_c[2] += self.Te[2][3]

        self.x_cliped = np.clip(self.x_c, self.min_x, self.max_x)
        print("\nx_cliped: ", self.x_cliped)

        self.og_Te[0][3] = self.x_cliped[0]
        self.og_Te[1][3] = self.x_cliped[1]
        self.og_Te[2][3] = self.x_cliped[2]

        # Calculate joint positions
        self.sol = self.robot.ikine_LM(SE3(self.og_Te), q0 = self.q0)
        self.point.positions = self.sol.q

        # Publish UR5 velocity and position
        self.unpause()
        self.point.time_from_start = rospy.Duration(self.t)
        self.q_cmd.points.append(self.point)
        self.ur_cmd.publish(self.q_cmd)
        time.sleep(0.5)
        # time.sleep(1.5)
        self.pause()

        self.new_obs = self.get_observation()
        self.reward, self.done, self.info = self.calculate_reward(self.
    new_obs)
```

```python

        # self.info = None

        return self.new_obs, self.reward, self.done, self.info
```

Listing 5: Variable PD Lifting Environment

Appendix F Fixed Impedance Lifting Environment

```

#!/usr/bin/env python

import numpy as np

# Torch imports
from torch.utils.tensorboard import SummaryWriter

# Robot and task space import
from ur_imp_lift import UR_IMP_LIFT
from ur_imp_reach import UR_IMP_REACH

#Select Env and comment the other
env = UR_IMP_LIFT()
env = UR_IMP_REACH()
max_eps = 450
max_steps = 50
total_timesteps = 0

writer = SummaryWriter(comment="TD3_FixdIMP_reach_4.8")

for eps in range(max_eps):

    state =  env.reset()
    episode_reward = 0
    rewards = []
    done = False

    for step in range(max_steps):
```

```python
    # while not done:
        action = np.array([4.8, 4.8, 4.8])
        next_state, reward, done, info = env.step(action)
        total_timesteps += 1
        episode_reward += reward
        state=next_state
        writer.add_scalar("reward_step", reward, total_timesteps)

        if done:
            break

    rewards.append(episode_reward)
    avg_reward = np.mean(rewards[-100:])
    print("\nAvg_reward = ", avg_reward)
    writer.add_scalar("avg_reward", avg_reward, total_timesteps)
    writer.add_scalar("episode_reward", episode_reward, eps)
    writer.add_scalar("Difference in x", info[0], eps)
    writer.add_scalar("Difference in y", info[1], eps)
    writer.add_scalar("Difference in z", info[2], eps)

    print('Episode: ', eps, '| Episode Reward: ', episode_reward)
```

Listing 6: Fixed Impedance Lifting Environment

Bibliography

[1] Hybrid position/force control, velocity projection, and passivity. *IFAC Proceedings Volumes*, 30(20):325–331, 1997. 5th IFAC Symposium on Robot Control 1997 (SYROCO '97), Nantes, France, 3-5 September.

[2] Fares J. Abu-Dakka, Leonel Rozo, and Darwin G. Caldwell. Force-based learning of variable impedance skills for robotic manipulation. In *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*, pages 1–9, 2018.

[3] Fares J. Abu-Dakka and Matteo Saveriano. Variable impedance control and learning—a review. *Frontiers in Robotics and AI*, 7, 2020.

[4] A. Albu-Schaffer and G. Hirzinger. Cartesian impedance control techniques for torque controlled light-weight robots. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 1, pages 657–663 vol.1, 2002.

[5] Akhil S. Anand, Rituraj Kaushik, Jan Tommy Gravdahl, and Fares J. Abu-Dakka. Data-efficient reinforcement learning for variable impedance control. *IEEE Access*, 12:15631–15641, 2024.

[6] Rika Antonova, Silvia Cruciani, Christian Smith, and Danica Kragic. Reinforcement learning for pivoting task. 03 2017.

[7] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[8] Aude Billard and Danica Kragic. Trends and challenges in robot manipulation. *Science*, 364(6446):eaat8414, 2019.

[9] Jonathan Bohren, Radu Bogdan Rusu, E. Gil Jones, Eitan Marder-Eppstein, Caroline Pantofaru, Melonee Wise, Lorenz Mösenlechner, Wim Meeussen, and Stefan Holzer. Towards autonomous robotic butlers: Lessons learned with the pr2. In *2011 IEEE International Conference on Robotics and Automation*, pages 5568–5575, 2011.

[10] Jonas Buchli, Evangelos Theodorou, Freek Stulp, and Stefan Schaal. Variable impedance control a reinforcement learning approach. 07 2010.

[11] Fabrizio Caccavale, Pasquale Chiacchio, Alessandro Marino, and Luigi Villani. Six-dof impedance control of dual-arm cooperative manipulators. *IEEE/ASME Transactions on Mechatronics*, 13(5):576–586, 2008.

[12] Chien-Chern Cheah and Danwei Wang. Learning impedance control for robotic manipulators. *IEEE Transactions on Robotics and Automation*, 14(3):452–465, 1998.

[13] Adam Coates and Andrew Y. Ng. Multi-camera object detection for robotics. In *2010 IEEE International Conference on Robotics and Automation*, pages 412–419, 2010.

[14] J.J. Craig and M.H. Raibert. A systematic method of hybrid position/force control of a manipulator. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.*, pages 446–451, 1979.

[15] Zihan Ding. Popular-rl-algorithms. `https://github.com/quantumiracle/Popular-RL-Algorithms`, 2019.

[16] Neel Doshi, Orion Taylor, and Alberto Rodriguez. Manipulation of unknown objects via contact configuration regulation. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 2693–2699, 2022.

[17] Thomas Eiband, Matteo Saveriano, and Dongheui Lee. Learning haptic exploration schemes for adaptive task execution. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7048–7054, 2019.

[18] Eric L. Faulring, Kevin M. Lynch, J. Edward Colgate, and Michael A. Peshkin. Haptic display of constrained dynamic systems via admittance displays. *IEEE Transactions on Robotics*, 23(1):101–111, 2007.

[19] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[20] Ali Ghadirzadeh, Atsuto Maki, Danica Kragic, and Mårten Björkman. Deep predictive policy training using reinforcement learning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2351–2358, 2017.

[21] Michael A. Goodrich and Alan C. Schultz. 2008.

[22] Stavros Grafakos, Fotios Dimeas, and Nikos Aspragathos. Variable admittance control in phri using emg-based arm muscles co-activation. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 001900–001905, 2016.

[23] B. Heinrichs, N. Sepehri, and A.B. Thornton-Trump. Position-based impedance control of an industrial hydraulic manipulator. *IEEE Control Systems Magazine*, 17(1):46–52, 1997.

[24] Neville Hogan. Impedance control: An approach to manipulation. In *1984 American Control Conference*, pages 304–313, 1984.

[25] Guanhua Hu, Qingjiu Huang, and Takuya Hanafusa. Hybrid position/force control with virtual impedance model of robot manipulators. In *Journal of Physics: Conference Series*, volume 1601, page 062014. IOP Publishing, 2020.

[26] R. Ikeura, T. Moriguchi, and K. Mizutani. Optimal variable impedance control for a robot and its application to lifting an object with a human. In *Proceedings. 11th IEEE International Workshop on Robot and Human Interactive Communication*, pages 500–505, 2002.

[27] Alireza Izadbakhsh and Saeed Khorashadizadeh. Robust impedance control of robot manipulators using differential equations as universal approximator. *International Journal of Control*, 91(10):2170–2186, 2018.

[28] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[29] Gitae Kang, Hyun Seok Oh, Joon Kyue Seo, Uikyum Kim, and Hyouk Ryeol Choi. Variable admittance control of robot manipulators based on human intention. *IEEE/ASME Transactions on Mechatronics*, 24(3):1023–1032, 2019.

[30] Parham M. Kebria, Saba Al-wais, Hamid Abdi, and Saeid Nahavandi. Kinematic and dynamic modelling of ur5 manipulator. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 004229–004234, 2016.

[31] Arvid QL Keemink, Herman van der Kooij, and Arno HA Stienen. Admittance control for physical human–robot interaction. *The International Journal of Robotics Research*, 37(11):1421–1444, 2018.

[32] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[33] P Lammertse. Admittance control and impedance control-a dual. *FCS Control Systems*, 13, 2004.

[34] Vincenzo Lippiello, Bruno Siciliano, and Luigi Villani. A position-based visual impedance control for robot manipulators. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 2068–2073, 2007.

[35] Jianlan Luo, Eugen Solowjow, Chengtao Wen, Juan Aparicio Ojea, Alice M. Agogino, Aviv Tamar, and Pieter Abbeel. Reinforcement learning on variable impedance controller for high-precision robotic assembly, 2019.

[36] Shan Luo, Joao Bimbo, Ravinder Dahiya, and Hongbin Liu. Robotic tactile perception of object properties: A review. *Mechatronics*, 48:54–67, 2017.

[37] Rosasco L. Maiettini E., Pasquale G. and Natale L. On-line object detection: a robotics challenge. *Autonomous Robots*, 44:739–757, 2020.

[38] J Maples and Joseph Becker. Experiments in force control of robotic manipulators. In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, volume 3, pages 695–702. IEEE, 1986.

[39] Roberto Martín-Martín, Michelle A. Lee, Rachel Gardner, Silvio Savarese, Jeannette Bohg, and Animesh Garg. Variable impedance control in end-effector space: An action space for reinforcement learning in contact-rich tasks. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1010–1017, 2019.

[40] Allison M. Okamura and Mark R. Cutkosky. Feature detection for haptic exploration with robotic fingers. *The International Journal of Robotics Research*, 20(12):925–938, 2001.

[41] M. H. Raibert and J. J. Craig. Hybrid Position/Force Control of Manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 103(2):126–133, 06 1981.

[42] Maurizio Valle Ravinder S. Dahiya. *Robotic Tactile Sensing*. Springer Dordrecht, 2012.

[43] Mario Richtsfeld and Markus Vincze. Grasping of unknown objects from a table top. In *Workshop on Vision in Action: Efficient strategies for cognitive agents in complex environments*, Marseille, France, October 2008. Markus Vincze and Danica Kragic and Darius Burschka and Antonis Argyros.

[44] Rocco A. Romeo and Loredana Zollo. Methods and sensors for slip detection in robotics: A survey. *IEEE Access*, 8:73027–73050, 2020.

[45] Leonel Rozo, Sylvain Calinon, Darwin Caldwell, Pablo Jimenez, and Carme Torras. Learning collaborative impedance-based robot behaviors. 07 2013.

[46] Behzad Sadrfaridpour, Maziar Fooladi Mahani, Zhanrui Liao, and Yue Wang. Trust-based impedance control strategy for human-robot cooperative manipulation. page V001T04A015, 09 2018.

[47] S.A. Schneider and R.H. Cannon. Object impedance control for cooperative manipulation: theory and experimental results. *IEEE Transactions on Robotics and Automation*, 8(3):383–394, 1992.

[48] Jian Shi, J. Zachary Woodruff, Paul B. Umbanhowar, and Kevin M. Lynch. Dynamic in-hand sliding manipulation. *IEEE Transactions on Robotics*, 33(4):778–795, 2017.

[49] Bruno Siciliano and Luigi Villani. An inverse kinematics algorithm for interaction control of a flexible arm with a compliant surface. *Control Engineering Practice*, 9(2):191–198, 2001.

[50] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

[51] Mark W Spong, Frank L Lewis, and Chaouki T Abdallah. *Robot control: dynamics, motion planning, and analysis*. IEEE press, 1992.

[52] Alexander L. Strehl, Lihong Li, Eric Wiewiora, John Langford, and Michael L. Littman. Pac model-free reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 881–888, New York, NY, USA, 2006. Association for Computing Machinery.

[53] Taisuke Sugaiwa, Genki Fujii, Hiroyasu Iwata, and Shigeki Sugano. A methodology for setting grasping force for picking up an object with unknown weight, friction, and stiffness. In *2010 10th IEEE-RAS International Conference on Humanoid Robots*, pages 288–293, 2010.

[54] Sonny Tarbouriech, Benjamin Navarro, Philippe Fraisse, André Crosnier, Andrea Cherubini, and Damien Sallé. Admittance control for collaborative dual-arm manipulation. In *2019 19th International Conference on Advanced Robotics (ICAR)*, pages 198–204, 2019.

[55] Dzmitry Tsetserukou, Naoki Kawakami, and Susumu Tachi. isora: Humanoid robot arm for intelligent haptic interaction with the environment. *Advanced Robotics*, 23:1327–1358, 01 2009.

[56] Luigi Villani and Joris De Schutter. *Force Control*, pages 195–220. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[57] Yanjun Wang. *Impedance control without force sensors with application in homecare robotics*. PhD thesis, University of British Columbia, 2014.

[58] Yangsheng Xu, Richard P. Paul, and Peter I. Corke. Hybrid position force control of robot manipulator with an instrumented compliant wrist. In Vincent Hayward and Oussama Khatib, editors, *Experimental Robotics I*, pages 244–270, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[59] Chenguang Yang, Guangzhu Peng, Yanan Li, Rongxin Cui, Long Cheng, and Zhijun Li. Neural networks enhanced adaptive admittance control of optimized robot–environment interaction. *IEEE Transactions on Cybernetics*, 49(7):2568–2579, 2019.

[60] T. Yoshikawa. Dynamic hybrid position/force control of robot manipulators–description of hand constraints and calculation of joint driving force. *IEEE Journal on Robotics and Automation*, 3(5):386–392, 1987.

[61] T. Yoshikawa and A. Sudou. Dynamic hybrid position/force control of robot manipulators-on-line estimation of unknown constraint. *IEEE Transactions on Robotics and Automation*, 9(2):220–226, 1993.

[62] T. Yoshikawa, T. Sugie, and M. Tanaka. Dynamic hybrid position/force control of robot manipulators-controller design and experiment. *IEEE Journal on Robotics and Automation*, 4(6):699–705, 1988.

[63] Ganwen Zeng and Ahmad Hemami. An overview of robot force control. *Robotica*, 15(5):473–482, 1997.