MONTAN UNIVERSITÄT LEOBEN

Lehrstuhl für Cyber Physical Systems

Masterarbeit

A Motor Control Learning Framework for Cyber-Physical-Systems

Nikolaus Feith, BSc

Juni 2022

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 30.05.2022

Unterschrift Verfasser/in
Nikolaus Feith

ABSTRACT

A central problem in robotics is the description of the movement of a robot. This task is complex, especially for robots with high degrees of freedom. In the case of complex movements, they can no longer be programmed manually. Instead, they are taught to the robot utilizing machine learning. The Motor Control Learning framework presents an easy-to-use method for generating complex trajectories. Dynamic Movement Primitives is a method for describing movements as a non-linear dynamic system. Here, the trajectories are modelled by weighted basis functions, whereby the machine learning algorithms must determine only the respective weights. Thus, it is possible for complex movements to be defined by a few parameters. As a result, two motion learning methods were implemented. When imitating motion demonstrations, the weights are determined using regression methods. A reinforcement learning algorithm is used for policy optimization to generate waypoint trajectories. For this purpose, the weights are improved iteratively through a cost function using the covariance matrix adaptation evolution strategy. The generated trajectories were evaluated in experiments.

KURZFASSUNG

Ein zentrales Problem in der Robotik ist die Beschreibung der Bewegung eines Roboters. Diese Aufgabe ist komplex, insbesondere bei Robotern mit hohen Freiheitsgraden. Bei komplexen Bewegungen können diese nicht mehr manuell programmiert werden. Stattdessen werden sie dem Roboter mit Hilfe von maschinellem Lernen beigebracht. Das Motor Control Learning Framework stellt eine einfach zu bedienende Methode zur Erzeugung komplexer Trajektorien dar. Dynamic Movement Primitives ist eine Methode zur Beschreibung von Bewegungen als nichtlineares dynamisches System. Dabei werden die Trajektorien durch gewichtete Basisfunktionen modelliert, wobei die maschinellen Lernalgorithmen nur die jeweiligen Gewichte bestimmen müssen. So ist es möglich, dass komplexe Bewegungen durch wenige Parameter definiert werden können. Als Ergebnis wurden zwei Bewegungslernverfahren implementiert. Bei der Nachahmung von Bewegungsdemonstrationen werden die Gewichte mittels Regressionsverfahren bestimmt. Für die Optimierung der Policy zur Generierung von Wegpunkt-Trajektorien wird ein Reinforcement-Learning-Algorithmus verwendet. Zu diesem Zweck werden die Gewichte iterativ durch eine Kostenfunktion unter Verwendung der Covariance Matrix Adaptation Evolution Strategy verbessert. Die generierten Trajektorien wurden in Experimenten evaluiert.

Contents

List of Figures

List of Tables

1 Introduction

Robots are highly complex systems that involve many sub-disciplines of engineering. Working effectively with these systems requires a solid base of knowledge and a good development environment that takes away the cumbersome preliminary work. This thesis aims to solve these problems and, on the one hand, to give an introduction to the basics like kinematics and robot control.

Therefore, the purpose of this thesis is first to provide an introduction to robotics and the application of machine learning to robot control. Subsequently, a framework was programmed that allows the generation of trajectories for the movement of robots. Two learning methods were used for this purpose. The first one uses reinforcement learning to generate waypoint trajectories. For this purpose, the method of Rueckert and d'Avella (2013) was implemented, which applies policy search to dynamic movement primitives. Therefore, the Covariance Matrix Adaption evolution strategy is used to optimize the policy. For the application, only the via points and the temporal scaling of the trajectory has to be specified. All other parameters are used for the motion function's resolution or oscillation properties.

The second application of the framework is the imitation of motions presented by demonstrations. These demonstrations can be performed either by the reinforcement learning algorithm or manually by an instructor using the robot. Here again, Dynamic Movement Primitives are used to model the movements. However, the motions are not altered by reinforcement learning but by ridge regression. Furthermore, the method of Paraschos et al. (2018) for jerk optimization for movement primitives was implemented, which allows a further improvement of the learned movements.

Finally, use cases for the framework will be presented; these will show the application in teaching, industry and research. These cases should give an outlook on the versatility of the motor control learning framework.

1.1 Motivation

In recent years, robots outside industrial plants and research facilities have become more common. In the next few years, this trend will increase even further. In particular, the development of intelligent and autonomous robotic systems represents a particular challenge for this decade. Thus, a central task of robotics is how to teach complex tasks to robots as simply as possible. For this purpose, a wide variety of machine learning algorithms have been developed, whereby the field of reinforcement learning, in particular, plays a significant role.

Furthermore, simulations are becoming more and more critical as they provide engineers and developers with a comparatively cheap, risk-free and time-efficient method for evaluating motion sequences. In order to use this advantage in reinforcement learning, methods for generating trajectories, such as dynamic movement primitives, have been developed. These movement representations provide a flexible and smart solution for countless tasks. Finally, intelligent frameworks for robots, such as Robot Operating System (ROS), enable controllers to be developed easily and quickly and allow fast switching between real and simulated environments.

Nevertheless, simulation and the possibility of controlling real robots is an crucial task for the Chair of Cyber-Physical Systems. It is essential that a very general control framework can be used, which functions independently of the platform. Especially at the chair, there are three different serial robots as of May 2022, which should all be controlled with the same motor control learning framework. These include robots from Fanuc, Universal Robotics and Franka Emika.

(a) *Fanuc CRX-10iA* **(b)** *UR3* **(c)** *Panda*

Figure 1: *This figure shows the three robots of the Chair of Cyber-Physical Systems.*

From these issues, i.e. the control of both simulated and real robots, the framework's design follows. First, the possibility to generate smooth trajectories for an undefined number of waypoints shall be created. Then, the user should define the dimensionality of the trajectories. For the task space, this would be three dimensions for the location or six for the position, including the orientation. Nevertheless, the application of joint space trajectories, which have seven dimensions in the case of the Panda robot and six dimensions in the case of the CRX-10iA, as well as UR3, should be possible.

The second application is the imitation of demonstrations. For this application, the transferability of movements between robots is particularly interesting through task space trajectories. In particular, by using Dynamic Movement Primitives, these learned movements can be scaled in time and space, and the target positions can be changed. Thereby, the usage of various robots can be further increased. Therefore, the motivation of this thesis is to bundle these learning methods into a framework that is easy to use. Furthermore, the motor control framework should have three use cases: teaching, industry, and research.

1.2 Related Work

At the beginning of the development of a trajectory generator, critical key points have to be decided at the beginning. For this purpose, the decision characteristics for a model-based and model-free method are listed below. The reinforcement learning and the optimizer have to be adapted to each other. However, the transfer of simulated systems to real systems is a challenge. It is essential here to close the so-called reality gap.

1.2.1 Types of Reinforcement Learning

In general, reinforcement learning (RL) is divided into model-based and model-free RL. As the name suggests, model-based RLs know the environment in which the agent operates. Either the model of the environment is given, or the algorithm's goal is to learn its model. Meaning that the agent knows the state transition and searches for an optimal policy for the path from the current state to the target state. (Ravishankar and Vijayakumar, 2017)

On the other hand, model-free RLs do not know the transition model or reward function. This implies that the agent gains experience through trial and error and subsequently optimizes the policy with the help of the maximum reward. Furthermore, the method of Q-learning does not optimize the policy but improves the value function with the help of the Bellman equation.(Ravishankar and Vijayakumar, 2017; Ravichandiran, 2020)

Finally, Deep Reinforcement Learning extends the field of RLs through the implementation of artificial neural networks. These can be model-based or model-free and are used especially with high-dimensional data, for example, images or machines with many sensors, since conventional reinforcement learning cannot handle such large state spaces. (Arulkumaran et al., 2017; Ravichandiran, 2020)

1.2.2 Optimizer Types

In general, a distinction is made between white box and black-box optimization. In addition, there is the possibility of combining both, which is called a grey-box optimizer. A complete physical model of the problem in white-box optimization is known. With this model, first and second derivatives can be obtained, and the steepest path to the global optimum can be determined.(Vierhaus et al., 2017; Yang et al., 2017)

In contrast, a set of samples is generated randomly in black-box optimizers, depending on the method. Another way is to calculate according to an algorithm. Subsequently, the samples are used in a simulation of the problem. The result of the simulation is evaluated in an objective function. This step is repeated several times with different parameter sets. Only the result of the objective function is used in the further course to optimize the parameter generation. (Vierhaus et al., 2017)

1.2.3 Simulation vs Real Systems

Over the last two decades, simulations have become an essential tool in robotics. As Žlajpah (2008) describes, it enables, among other things, faster development times, generation of training data and the possibility of using non-existent resources as well as sparing expensive components. However, simulations have one central problem despite these and many other advantages. They only represent reality to a limited extent. This discrepancy between simulation and reality is named the reality gap as it is called in Bousmalis and Levine (2017) and Mouret and Chatzilygeroudis (2017) and is a challenge to all robot developers and researchers. The reality gap is caused by not perfectly representing reality, which distorts the simulation results. One way to close this gap is to develop better and better simulations, which are more expensive in computing power. Another is, as described in Mouret and Chatzilygeroudis (2017) and Koos, Mouret, and Doncieux (2013) the development of transferable controllers.

1.3 Learning Methods

In this thesis, two learning methods for Motor Control learning are described. In Imitation Learning (IM), the agent, a robot, learns skills or activities through demonstrations given by a teacher. The teacher can be a human or a data source like a video. These demonstrations can then be learned as Dynamic Movement Primitives (DMPs) with Schaal et al. (2003) using a regression model. This method was extended by Paraschos et al. (2018), so that not only the demonstrations can be imitated, but also the jerk of the motor control can be minimized.

The second learning method is an application of reinforcement learning (RL). For this purpose, motor controls are again modelled as DMPs, and subsequently, via-point motor controls can be learned using policy optimizer methods. For this purpose, the method of Rueckert and d'Avella (2013) is applied, which uses a Black Box Optimizer (BBO) called Covariance Matrix Adaption Evolution Strategy (CMA-ES), which was first presented in Hansen and Ostermeier (2001), to learn the weights of the DMPs.

1.4 Use Cases

The motor framework is supposed to have three use cases described in the following. These are intended to facilitate the work of the Chair of Cyber-Physical Systems with the Panda robotic arm from the company Franka Emika GmbH. The areas of teaching, industry and research were chosen as applications.

1.4.1 Teaching

The teaching use case should combine two areas, which will be described below. The central idea is to simplify teaching the complex field of robotics and the application of machine learning, including simulations, robot controls, and communication interfaces, like ZMQ and Robot Operating System (ROS). The simulation program chosen for this purpose is called "CoppeliaSim".

The first access is intended to introduce the simulation of the Franka robot. For example, the content of the course can be the implementation of forwarding and inverse kinematics and the programming of a Jacobian inverse controller. The communication in this application runs over ZMQ, an asynchronous message library. The sophisticated approach uses ROS for communication and control. This method allows for much more flexible applications, especially the possibility of testing the programs on a real robot. Additionally, with this approach, there is no binding to CoppeliaSim, and other simulation programs like Gazebo can be used.

In both cases, the motor framework is used to generate trajectories created either by reinforcement learning or imitation learning. In addition, the framework is intended to provide an easy way to generate demonstrations for teaching purposes. It will also give students a basis for their bachelor's and master's theses so that they can dive deeper into robotics and do not have to deal with the control of serial robot arms.

1.4.2 Industry

In the "Industry" use case, the handling of the real robot is made more convenient for the user. Furthermore, the framework offers the possibility to define movements utilizing waypoints or to train them with the help of imitation learning. This use of the robot arm is intended to be particularly simple to help simplify the work process in industrial cases. Furthermore, this application should allow the Chair of Cyber-Physical Systems to present recent developments to industrial partners.

1.4.3 Research

Finally, the "Research" use case is intended to provide researchers with a solid foundation for developing new methods for the Franka Emika Panda robotic arm. It is not meant to be a constraining one but to provide the opportunity for modifications and extensions. Furthermore, it is kept so general that even if the robot is changed, many functionalities can be used for the different robot, and therefore an entirely new framework does not have to be created. Only the robot controller has to be changed to compensate for the variation in dynamics.

1.5 Outlook

In this thesis, first an introduction to the required methods of robotics is given in **Chapter 2** and machine learning in **Chapter 3**. Subsequently, the software components and the robot are described in **Chapter 4**. In **Chapter 5**, the conducted experiments are presented and their results. Finally, a conclusion is drawn, the results are discussed, and further work is described.

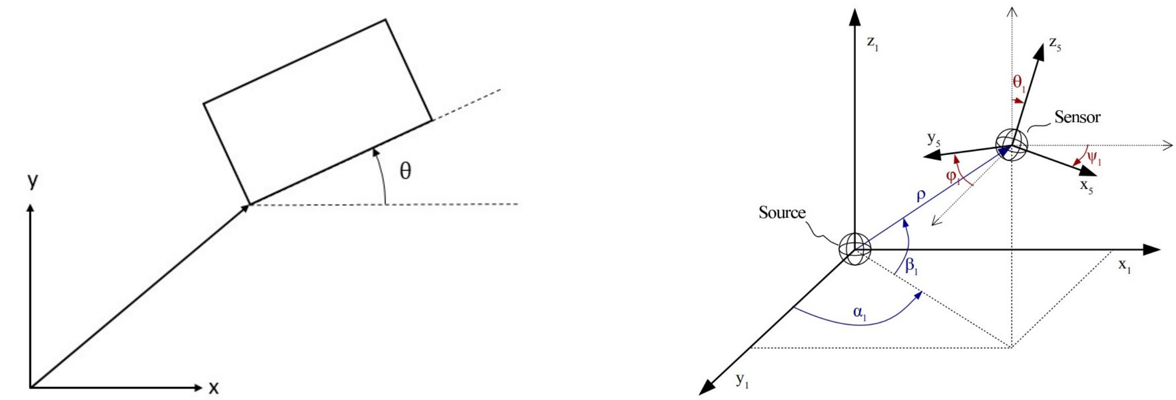
2 Background Methods in Robotics

This chapter introduces the fundamental areas of robotics needed to understand and apply the CPS framework. For this purpose, essential terms of robotics, such as degrees of freedom or kinematic chains, are explained in the first section. The second section gives an introduction to the mathematical methods, as well as forward and inverse kinematics. Furthermore, the Denavit-Hartenberg parameters are introduced. In the last section, the Jacobian Inverse and Jacobian Transpose controllers are presented; these are powerful control algorithms in robotics.

2.1 Robot Basics

In this section, the basic concepts of robot manipulators are introduced. First, the idea of degrees of freedom and Grübler's formula is proposed. Then an overview of mathematical spaces in robotics and kinematic chains is given. Finally, as a typical example, a planar robot manipulator consisting of an open kinematic chain with two links is used in robotics.

2.1.1 Degrees of Freedom and the Grübler's Formular



(a) *planar* **(b)** *spatial*

Figure 2: *In these figures, the planar* ***(a)*** *and spatial* ***(b)*** *degrees of freedom are illustrated (Teixeira Silva et al., 2017)*

.

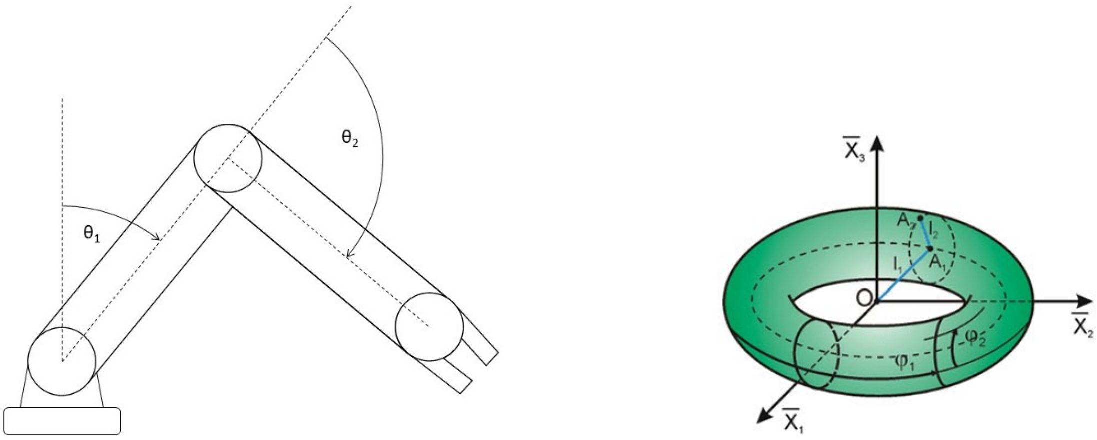
The number of degrees of freedom(DOF) is the number of independent variables needed to completely describe a mechanical system, e.g. a robot, and all possible configurations. A configuration is the physical state of the robot, especially of the joints, concerning its environment, more about this in the next paragraph. For example, a water tap has only one degree of freedom, the state of the valve, which regulates the flow of water. For planar movement, there are two degrees of freedom, (X, Y), for motions without orientations and three DOFs, (X, Y,θ), for movements with orientations which are shown in Figure 2a. Spatial Movements need six independent coordinates, as illustrated in Figure 5b, to describe an unique position, (X,Y,Z), and orientation, (ϕ, θ, ψ). There are many conventions to describe an orientation in space; in this thesis, the convention of Euler angles is generally used (Lynch and Park, 2017; Dudek and Jenkin, 2010).

$$DOF = m(N-1) - \sum_{i=1}^{J}(m - f_i). \tag{1}$$

Grübler's formula, Equation (1), is a method to determine the degrees of freedom of a mechanical system. For this purpose, the number of rigid bodies N and joints J and their degrees of freedom(DOFs of a rigid body m and a joint f_i) is specified. As a result of Grübler's formula, the number of independent variables, the DOFs, of the system is obtained. It is essential whether one is in a plane or spatial system because the degrees of freedom of rigid bodies in space and the plane differ, as already stated, by 3 degrees of freedom (Lynch and Park, 2017).

2.1.2 Configuration Space

The configuration space (C-Space) is a mathematical-topological space in which every possible joint state of a mechanical mechanism, in this case, a robot manipulator, can be represented. The state coordinates are thereby specified in generalized form. Thus, the space has exactly the number of independent variables as the observed system has degrees of freedom and the position and orientation in space(which is generally omitted for fixed-mounted robots). In C-Space, the joint spaces are equivalent to their actual characteristics. Especially for revolute joints, this property is important because the states 0 and 2π are glued together and are continuous. A configuration q denotes a unique state of the robot. An example of this is a two-link robot shown in Figure 3a, where a torus describes its configuration space, Figure 3b, as ϕ_1 and ϕ_2 are revolute joints (Lynch and Park, 2017; Kelly, Davila, and Perez, 2006).



(a) *Two link Robot* **(b)** *Configuration Space*

Figure 3: ***(a)*** *shows a two link robot and* ***(b)*** *its configuration space is presented (Lynch and Park, 2017).*

2.1.3 Task Space and Work Space

The task space and the workspace both do not describe the whole robot but the configuration of the end effector. These spaces can be Cartesian spaces (in the most common cases) or other coordinate systems, which are more suitable for the description of the robot's end-effector motions (Lynch and Park, 2017).

The task space refers to the space in which a task is performed. Thus, the description depends only on the action to be performed and not on the robot. For this purpose, a coordinate system is used that best suits the task, so if, for example, if a picture is to be drawn, the $\mathbb{R}^2$ is used because the drawing only exists in the plane (Lynch and Park, 2017).

On the other hand, the workspace describes the end-effector position concerning the robot's configuration. The workspace describes the configurations of the end effector and includes the knowledge of the joint limits and is therefore independent of the tasks that the robot has to perform. Note that depending on the structure of the robot and its joint boundaries, certain positions of the end effector

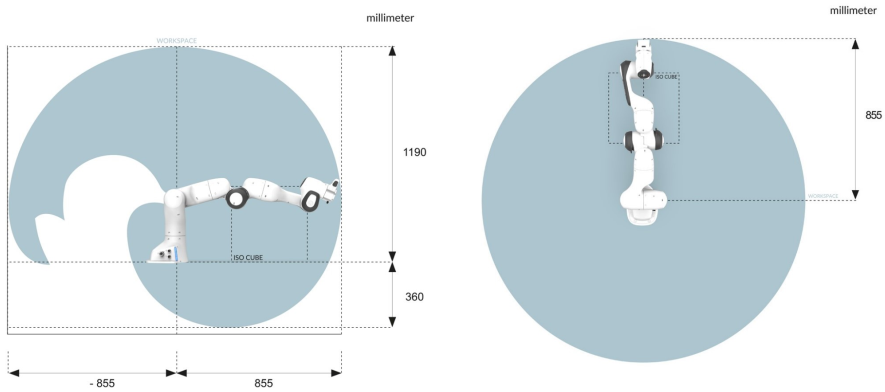


Figure 4: *These figures show the workspace in side view, left, and top view, right, of the Franka robot (Franka-Emika-GmbH, 2018).*

are not reachable or are reachable through several configurations; see Figure 4. This issue can lead to singularities, i.e., the transitions between two end effector positions are small, but the two resulting configurations are too far apart. As a result, the joint velocities become infinitely large, which could damage the robot, or it is not possible at all. These problems can be compensated by additional joints (Lynch and Park, 2017; Mareczek, 2020).

2.1.4 Kinematic Chain

Kinematic chains are assemblies consisting of links and joints located between the links. The joints generally have one degree of freedom, i.e. they are either revolute joints (R) or prismatic joints (P). Many other joints, such as screw joints (H) or universal joints (U), are either rarely used in serial robots or only in parallel robots and are therefore not of interest to this thesis. Furthermore, kinematic chains can be distinguished between open-chain and closed-chain mechanisms. Since this thesis only deals with the Franka Emika Panda and its application, which is a serial robot and therefore an open kinematic chain (Constans and Dyer, 2018).

Open kinematic chains are all those mechanisms where the end effector is connected to the chain with only one side. An example of this would be the human arm. In contrast, our two legs with the body (in this case, the end effector) would be a closed kinematic chain if both feet are fixed to the ground (Mareczek, 2020).

2.2 Kinematics and Dynamics

This section gives the essential elements of kinematics and dynamics for robotic manipulators, beginning with an introduction to planar and spatial coordinate transformations. Subsequently, the forward and inverse kinematics are formulated. Finally, the DH parameters are presented, a powerful method for constructing transformation matrices. However, first, the difference between kinematics and dynamics should be explained. Kinematics is the study of motion, considering it independent of forces and moments, indicating that movements are considered purely geometrically. The variables used here are position, velocity and acceleration. In contrast, dynamics describes motions due to changes in forces and moments (Mahnken, 2011).

2.2.1 Mathematical Methods for Robotics

In order to be able to describe robot movements, a method from multi-body dynamics, the coordinate transformation, is used. A distinction must be made between translations and rotations, and the axis of rotation is another critical parameter. In the following, first the equations for the planar and then the spatial movements are described. Finally, all equations are given in their generalized form (Lynch and Park, 2017).

Planar Transformations The following transformation describes the planar rotation (2) and the planar translation (3). Here $(x, y, 1)^T$ indicates the input vector, θ the rotation angle and a the shift in x-direction and b in y-direction. $(\hat{x}, \hat{y}, 1)^T$ denotes the transformed vector.

$$\text{Rot}(\theta) = \begin{pmatrix} \hat{x} \\ \hat{y} \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{2}$$

$$\text{Trans}(a, b) = \begin{pmatrix} \hat{x} \\ \hat{y} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{3}$$

Spatial Transformations Here the spatial transformations are described, starting with the spatial rotations around the x-axis, (4), the y-axis,(5), and the z-axis,(6), furthermore the spatial translation is given in Equation (7).

$$\text{Rot}(x, \theta) = \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \tag{4}$$

$$\text{Rot}(y, \phi) = \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \tag{5}$$

$$\text{Rot}(y, \psi) = \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \tag{6}$$

$$\text{Trans}(a, b, c) = \begin{pmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \tag{7}$$

In the equations (4) - (7), $(x, y, z, 1)^T$ denotes the input vector, $(\phi, \theta, \psi)^T$ denotes the angles of rotation about the x, y, z-axis, and $(a, b, c)^T$ denotes the displacement along these axes. The output of these transformations is the vector $(\hat{x}, \hat{y}, \hat{z}, 1)^T$.

2.2.2 Forward Kinematics

Forward kinematics or direct kinematics refers to the computation of the position and orientation of the end effector to the coordinate system of the robot base. For this purpose, a separate reference system is introduced for each link. The goal is to determine the state of the end effector given joint variables q_i, joint angle θ_i and joint displacements d_i. Mathematically, the problem is described in the following way:

$$\mathbf{x} = f_{fwd\ kin}(\mathbf{q}). \tag{8}$$

The forward problem is mapping the joint states to an end effector position and orientation. As a consequence, each configuration $\mathbf{q}$ is unique and generates an end-effector state $\mathbf{x}$. In the case of a serial robot with more than six DOFs, the end effector state is reachable from more than one configuration (Mareczek, 2020).

2.2.3 Inverse Kinematics

Inverse kinematics refers to the inverse problem of direct kinematics. It searches for the joint variables, angles and displacements, which result in a given position of the end effector, i.e., the position and orientation. The difficulty here is that inverse kinematics is a highly nonlinear problem for serial robots and, in addition, often does not produce unique solutions. Countable solutions denote finitely many singularities and uncountable solutions infinitely many. In addition, it may be that the desired position is outside the workspace. Therefore, inverse kinematics does not find a solution.

In the case of the Franka Emika Panda, it is more complex because the arm has 7 DOFs, whereby some singularities can be avoided. Because of the additional degrees of freedom, almost all end effector states can be reached by countless configurations. Therefore a method is required which reduces the number of possible configurations to a reasonable choice. Otherwise, a choice could be made, leading to infinitely large velocities in the transition from one state to the next. Because of the multiple solutions and the high nonlinearity, an analytical solution is usually not used for a higher number of DOFs, and numerical methods are used (Lynch and Park, 2017; Mareczek, 2020):

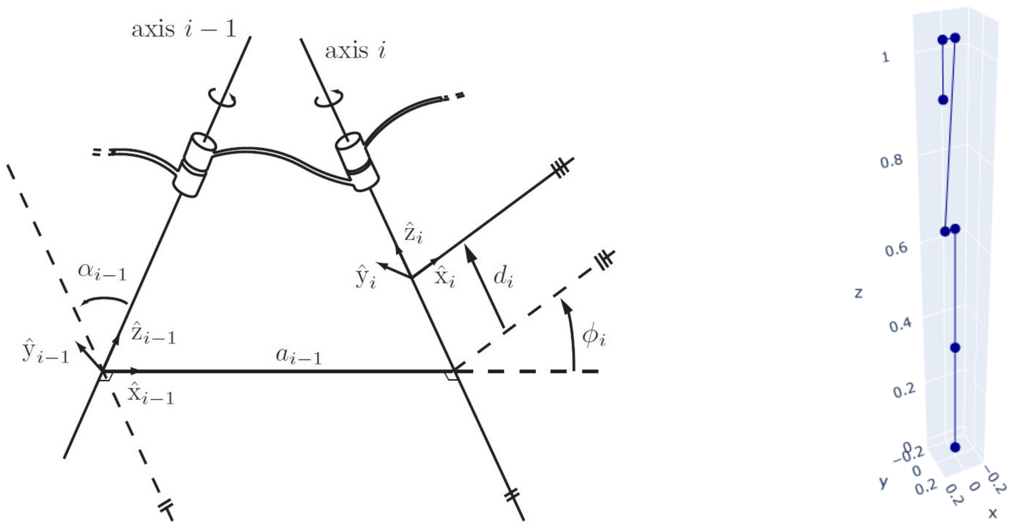
$$\mathbf{x} \underset{inv\ kin}{\mapsto} \mathbf{q}. \tag{9}$$

In order to apply inverse kinematics, a trajectory is required, which specifies the course of the end-effector's position. There are many ways to describe this, for example, a calculation by hand, trying it out with known functions, or it can be learned using Dynamic Movement Primitives, DMPs, and Reinforcement Learning, RL. More about the latter methods in the chapter 3. Subsequently, the trajectory is divided into smaller steps. Depending on the application, the step size can be fixed or variable. An essential factor here is how precisely the trajectory must be traversed. Then a control loop can be implemented, which has the target position as an input variable. This control loop is set up with the usage of the Jacobian matrix. More about it in the subsection Jacobian 2.3.1 (Niku, 2020).

2.2.4 Denavit-Hartenberg Parameters

The Denavit-Hartenberg parameter, DH, is an approach to finding a solution to the problem of forward kinematics. For this purpose, each robot manipulator is considered an open kinematic chain consisting of n links connected to joints with one degree of freedom. The problem of forward kinematics can now be described in the following way (10). Here {0} denotes the base frame and {n} the end-effector frame of the robot. Now the transformations $\mathbf{T}_{i-1,i}$ between the individual frames are set up. (Lynch and Park, 2017; Mareczek, 2020).

$$\mathbf{T}_{0,n}(\theta_1, ..., \theta_n) = \mathbf{T}_{0,1}(\theta_1), \mathbf{T}_{1,2}(\theta_2)...\mathbf{T}_{n-1,n}(\theta_n). \tag{10}$$



(a) *Denavit-Hartenberg Parameters*

(b) *Model of the Franka Robot Arm*

Figure 5: ***(a)*** *shows a graphical representation of Denavit-Hartenberg parameters (Lynch and Park, 2017) and* ***(b)*** *a model generated using the DH parameters from the Franka robot arm in its initial configuration.*

The transformations are now created as a combination of the Spatial Transformations (4)-(7) from the previous subsection 2.2.1.With this method, it is possible to reduce complex robot configurations to simpler models with only joints with one degree of freedom. For this purpose, the following parameters, the Denavit-Hartenberg parameters, are introduced, and an illustration of them is given in Figure 5a (Mareczek, 2020):

DH-Parameter	Name	Purpose
θ_i	joint angle	Indicates the angle which must be rotated around the joint axis z_{i-1} so that the axes x_{i-1} and x_i are oriented the same way. If the joint is prismatic, the θ_i remains constant. For revolute joints, the angle can be between $-\pi$ and π.
d_i	link offset	Denotes the distance between the intersection point between the origin {i-1} to the origin {i} along the axis z_{i-1}. Constant in the case of a rotational joint, variable in the case of prismatic joints.
a_i	link length	Denotes the distance between the joint axes z_{i-1} and z_i.
α_i	link twist	Denotes the twist of z_{i-1} and z_i with respect to x_{i-1} axis

Table 1: *This table lists the four Denavit-Hartenberg parameters (Mareczek, 2020).*

$$\begin{aligned}\mathbf{T}_{i-1,i} &= \text{Rot}(z, \theta_i)\ \text{Trans}(0, 0, d_i)\ \text{Trans}(a_i, 0, 0)\ \text{Rot}(x, \alpha_i), \\ &= \begin{pmatrix} \cos\theta_i & -\cos\alpha_i \sin\theta_i & \sin\alpha_i \sin\theta_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\alpha_i \cos\theta_i & -\sin\alpha_i \cos\theta_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}.\end{aligned} \tag{11}$$

After all the DH parameters of the robot have been set, the transformations can be constructed, and the concatenated transformation $\mathbf{T}_{0,n}$ can be computed. A small note about the Denavit-Hartenberg parameters, there are robot configurations where the parameters cannot be uniquely determined, so there are several sets of DH parameters, and it is important to stick to one description of the system. The individual transformations have the form of (11) (Mareczek, 2020).

2.3 Robot Control

A robot manipulator can perform an endless number of motions depending on the environment and the tasks the serial robot should perform. In all of these motions, control values of each joint must be continuously given to the robot's motors. Some control strategies have been established; in the following, two of these motion controls, the Jacobian Inverse Control and the Jacobian Transpose Control, will be presented. First, the calculation of the so-called Jacobian matrix is described in general and how to compute it with the help of the Denavit-Hartenberg parameters. Then, since the transformation matrices are not always square, a method is presented for how a non-square Jacobian matrix can be inverted (Lynch and Park, 2017).

2.3.1 Jacobian

The Jacobi matrix ,Equation (12), or functional matrix, denotes the derivative of a m-dimensional vector-valued function according to a n-dimensional argument vector. That means each component function f_i is derived after each argument x_i. The resulting matrix possesses dimension $m \times n$ (Gentle, 2017).

$$\begin{aligned}\frac{\partial \mathbf{f}}{\partial \mathbf{x}} &= \begin{bmatrix} \frac{\partial f_1}{\partial \mathbf{x}} & \frac{\partial f_2}{\partial \mathbf{x}} & \dots & \frac{\partial f_m}{\partial \mathbf{x}} \end{bmatrix}^{\mathbf{T}}, \\ &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_n} & \cdots & \frac{\partial f_2}{\partial x_2} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \end{aligned} \tag{12}$$

In order to use the Jacobian matrix for inverse or transpose control, the forward kinematics of the robot has to be set up first. In the last section, the Denavit-Hartenberg parameters 2.2.4 were presented for this purpose. The forward kinematics is a vector-valued function with dimension m, where m is the number of the end effector's degrees of freedom (pose and orientation). Furthermore, the argument vector has n dimensions, where n is the number of degrees of freedom, respectively the number of controllable joints. The functional derivative of the forward kinematics can be formed, which is the derivative of the transformation matrix according to the joint variables. In the following, the basic equations for spatial manipulators are shown (Lynch and Park, 2017).

Derivation of the Jacobian for serial robots:

$$\begin{aligned} \mathbf{x} &= f(\mathbf{q}), \\ \dot{\mathbf{x}} &= \frac{\mathrm{d}}{\mathrm{d}t} f(\mathbf{q}) = \frac{\mathrm{d}}{\mathrm{d}q} f(\mathbf{q}) \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{q} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}. \end{aligned} \tag{13}$$

Serial spatial Robot with DH-Parameters:

$$
\begin{aligned}
\mathbf{x}(\theta_1,\dots,\theta_n) &= \mathbf{T}_{0,n}(\theta_1,\dots,\theta_n),\\
\begin{bmatrix} x(\theta_1,\dots,\theta_n) \\ y(\theta_1,\dots,\theta_n) \\ z(\theta_1,\dots,\theta_n) \end{bmatrix} &= \begin{bmatrix} T_{0,n,x}(\theta_1,\dots,\theta_n) \\ T_{0,n,y}(\theta_1,\dots,\theta_n) \\ T_{0,n,z}(\theta_1,\dots,\theta_n) \end{bmatrix},\\
\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} &= \begin{bmatrix} \frac{\partial T_{0,n,x}}{\partial \theta_1} & \frac{\partial T_{0,n,x}}{\partial \theta_2} & \dots & \frac{\partial T_{0,n,x}}{\partial \theta_n} \\ \frac{\partial T_{0,n,y}}{\partial \theta_1} & \frac{\partial T_{0,n,y}}{\partial \theta_2} & \dots & \frac{\partial T_{0,n,y}}{\partial \theta_n} \\ \frac{\partial T_{0,n,z}}{\partial \theta_1} & \frac{\partial T_{0,n,z}}{\partial \theta_2} & \dots & \frac{\partial T_{0,n,z}}{\partial \theta_n}, \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \vdots \\ \dot{\theta}_n \end{bmatrix}.
\end{aligned}
$$

2.3.2 Jacobian Inverse Control

For this purpose, to apply Jacobian Inverse Control, the Jacobian matrix has to be inverted first, as the name of this method implies. For this purpose, it must be determined whether the matrix is invertible. If the determinant is not zero for square matrices, it is invertible. For non-square matrices, a pseudo inverse is calculated, which can be achieved if all rows or column vectors of the non-square matrix are linearly independent. The pseudo-inverse is most commonly calculated with the Moore-Penrose method, which is described as follows (14) - (15). The † symbol here only denotes that it is not a regular inverse matrix but a pseudo-inverse (Gentle, 2017; Lynch and Park, 2017).

$$\mathbf{J}^{\dagger} = \mathbf{J}^{\mathbf{T}}(\mathbf{J}\mathbf{J}^{\mathbf{T}})^{-1}, \qquad \text{if } \mathbf{J} \text{ is fat (n > m).} \tag{14}$$

$$\mathbf{J}^{\dagger} = (\mathbf{J}^{\mathbf{T}}\mathbf{J})^{-1}\mathbf{J}^{\mathbf{T}}, \qquad \text{if } \mathbf{J} \text{ is tall (n < m).} \tag{15}$$

Description of the Jacobian Inverse Equations:

$$\dot{\mathbf{q}} = \mathbf{J}^{-1}(\mathbf{q})\dot{\mathbf{x}}, \tag{16}$$

$$\Delta\mathbf{q} = \mathbf{J}^{-1}(\mathbf{q})\Delta\mathbf{x}, \tag{17}$$

$$\mathbf{q}\,[t+1] = \mathbf{q}\,[t] + \Delta\mathbf{q}, \tag{18}$$

$$= \mathbf{q}\,[t] + \eta\mathbf{J}^{-1}(\mathbf{q})\Delta\mathbf{x}. \tag{19}$$

Subsequently, the complete form of Jacobian inverse control can now be formulated. For this, the equation (13) is further transformed by inverting the Jacobian (16). Afterwards, the differential form is transformed into a different form, which means that the time steps are no longer continuous but discrete. This form is also called incremental form (17). Finally, a time step from t to $t+1$ is performed, equation (18), which describes the change of the joint angle, $\Delta\mathbf{q}$, for the next calculated position. The equation (19) gives the final form of the Jacobian inverse control. The η scales the step size from the end effector position $\mathbf{x}[t]$ to its target position $\mathbf{x}[t+1]$ (Craig, 2021).

2.3.3 Jacobian Transpose Control

Another control algorithm is the so-called Jacobian Transpose Control. The calculation method is much more efficient because no inverse has to be computed. Furthermore, trajectories which cross singularities are possible. In addition, with the help of the Jacobian Transpose torque control (21) can be applied. This type of control has the advantage that, on the one hand, the difference between the actual torques and the sensors in the revolute joints can be directly transferred to the motor control. Nevertheless, on the other hand, faster and more accurate configurations can be achieved. The formulas are provided in the following (Siciliano et al., 2008; Lynch and Park, 2017):

Jacobian Transpose Control for pose control:

$$\begin{aligned}
\dot{\mathbf{q}} &= \mathbf{J}^{\mathbf{T}}(\mathbf{q})\ \mathbf{e}, \\
\Delta\mathbf{q} &= \mathbf{J}^{\mathbf{T}}(\mathbf{q})\ (\mathbf{x}^{d} - f(\mathbf{q})), \\
\mathbf{q}\ [t+1] &= \mathbf{q}\ [t] + \eta\Delta\mathbf{q}, \\
&= \mathbf{q}\ [t] + \eta\mathbf{J}^{\mathbf{T}}(\mathbf{q})\ (\mathbf{x}^{d} - f(\mathbf{q})).
\end{aligned} \tag{20}$$

$$\tau = \mathbf{J}^{\mathbf{T}}(\mathbf{q})\mathcal{F}. \tag{21}$$

As with the equation (20), η denotes a scaling of the step size, $\mathbf{x}^{d}$ indicates the desired end-effector position, and $f(\mathbf{q})$ the current end-effector position. In torque control, the Jacobian transpose is multiplied by a force vector $\mathcal{F}$, and the result is the torques τ of the individual revolute joints. The application of force control is extremely complicated and will not be discussed further in this master thesis (Lynch and Park, 2017; Craig, 2021).

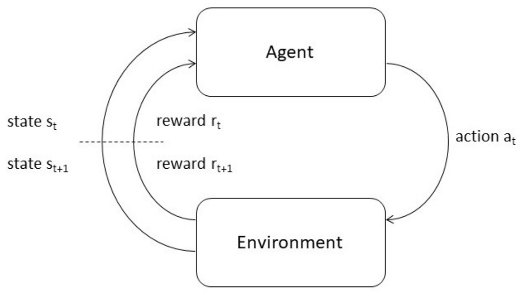
3 Background Methods in Machine Learning

This chapter first gives an overview of Machine Learning and then introduces Reinforcement Learning(RL). First, all the essential terms and concepts are described, and a mathematical description is provided. Subsequently, the covariance matrix adaptation evolution strategy (CMA-ES) is introduced and used in the CPS framework for policy optimization. Finally, an introduction to the Dynamic Movement Primitives (DMP) is given. These are the chosen model for the movement representations of this thesis.

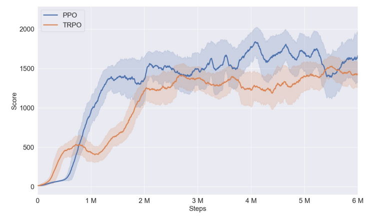
3.1 Reinforcement Learning (RL)

Reinforcement Learning (RL) is a sub-discipline of Machine Learning. In contrast to the other two areas, unsupervised and supervised machine learning, which are used for classification, clustering and regression, reinforcement learning is used to find a decision making policy which optimizes a given problem. Since the field of RL is very broad and includes countless algorithms. First, a rough introduction to the basic ideas of RL is given. Then it presents the fundamental elements and concepts and the Markov decision process (MDP). Finally, a black box optimizer, the CMA-ES, is introduced, which is used in the state of the art reinforcement learning algorithm (Ravichandiran, 2020).

3.1.1 Basic Idea and Fundamental Elements of Reinforcement Learning



(a) *Reinforcement learning*



(b) *Reward function*

Figure 6: ***(a)*** *shows a representation of reinforcement learning and* ***(b)*** *an example graph of a reward function (Lonza, 2019).*

The basic idea of reinforcement learning is to teach machines or computers a human learning method. For example, a stick is balanced upright for as long as possible. A child will be a bit clumsy at the beginning of this task, but after a short time, it will have understood the necessary knowledge about the stick dynamics to balance it, at least for a few seconds. The situation is similar to reinforcement learning, where some terms will now be introduced. In RL, the child would be the agent, and the stick and the physical world are the environments. A conceptional figure of an RL algorithm can be seen in Figure 6a. The agent can observe the states of the environment, more about this later, and influences these states by actions. The better the agent performs, the more reward it will receive from these actions. The individual components will be explained in more detail in the following subsections. In Figure 6b the progression of the reward of an RL example is shown.

Agent The agent, as previously mentioned, is one of the two entities in reinforcement learning. It observes the states of the environment and performs actions that affect it, by which it can get a reward. The agent is part of the reinforcement learning software and is supposed to solve the given problem more or less optimal. In particular, in model-free RL, the agent explores the state transitions or exploits its findings. Depending on the algorithm, the focus is either exploration or exploitation because an agent who knows only one way to the target state will probably not have found the optimal one. One who only explores may not find the target at all in the required time. This dilemma is called the exploration-exploitation dilemma (Ravichandiran, 2020; Lapan, 2020).

Environment Generally speaking, the environment is the agent's world, and the agent can only exist in it. The task of the environment is to process the interaction with the agent. As a result, it returns feedback to the agent in the form of the new state and the reward. In model-based RL, the environment is generally modeled as a Markov decision process (MDP), more on this in the paragraph 3.1.2 (Ravichandiran, 2020).

State and Observation To avoid ambiguity, RL distinguishes between states, s, and observations, o, of the environment by the agent. Observations are only a subset of the states measured by sensors, for example, so the agent does not have complete access to all states of the environment. Making the search for an optimal policy even more difficult, but it represents natural systems better since, in reality, one cannot or does not want to measure every state of a system. An example for states would be the tilt angle of the rod from the input example. However, this is not measured by the child directly but observed through its eyes (Ravichandiran, 2020; Bilgin, 2020).

Action Actions, a, are all those activities that the agent can perform in an environment and thereby change its states. These can be discrete or continuous, for example, the actions in a game of tic-tac-toe would be discrete since the agent can place its symbol in the 3x3 grid, but the movement of the hand while balancing a stick is continuous. The important thing is that the possible actions depend on the states of the environment (Ravichandiran, 2020).

Reward Finally, the term reward, r, is introduced. In reinforcement learning, the reward is a scalar quantity that indicates how well or poorly the agent behaves. Rewards can be positive or negative, and the magnitude is variable. Furthermore, the frequency with which rewards are distributed can be different. For example, dense reward functions punish or reward every action or sparse ones that only evaluate the game outcome at the end of each tic-tac-toe game. The design of the reward function is essential for RL because if the function offers loopholes to the algorithm, the agent may exploit them; this is called reward exploitation (Ravichandiran, 2020; Lonza, 2019).

3.1.2 Fundamental Concepts of RL

In this subsection, the fundamental concepts are added to the above definitions of reinforcement learning. In addition, the mathematical description of these concepts is also given here.

Action Space The action space denotes the set of all possible actions in the current environment. Furthermore, action spaces can be divided into discrete, for instance, the possible moves in a tic-tac-toe game or continuous action spaces. An example of continuous actions is the control of the robot joints or, to stay with the example of balancing a stick, the movement of the hand (Ravichandiran, 2020).

Policy The goal of using RL is to find a policy for an agent in an environment. A policy describes what action should be taken by the agent given a state or observation. Thereby, the expected cumulative reward should be maximal. Policies are divided into deterministic and stochastic ones. The deterministic, Equation 22, means that each state has only one possible action. Mathematically this is described in the following way:

$$a_t = f_\pi(s_t). \tag{22}$$

Here, the a_t denotes the action to be performed at time t, s_t denotes the state at that time, and f_π denotes the policy. The policy maps the current state to the action (Ravichandiran, 2020).

In contrast, the stochastic policy, Equation 24 does not assign an action to each state or observation but maps the action space with a probability distribution for each state. There are no limits on which probability distribution can be used. In the mathematical description, the mapping changes because it is no longer deterministic but stochastic, as follows (Ravichandiran, 2020):

$$a_t \sim \pi(a_t|s_t). \tag{23}$$

As an example of a stochastic policy, the Gauss policy is given here, where a state s is parameterized with the n-dimensional vector θ(Chou, Maturana, and Scherer, n.d.):

$$\begin{aligned} \pi_\theta(a|s) &= \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right), \\ with\ \mu &= \mu_\theta(s), \\ and\ \sigma &= \sigma_\theta(s). \end{aligned} \tag{24}$$

Furthermore, for discrete action spaces, categorical guidelines can be used which assign a frequency to each possible action. For example, in a grid world, the movement up, down, right or left is given a probability (up: 0.25, down: 0.1, right: 0.25, left: 0.4) (Ravichandiran, 2020).

Episode An episode denotes the transition of the agent from the initial state to a final state. The final state is optimally the goal state or a state that aborts the episode and punishes the agent for its mistake. The agent's path during an episode is called the trajectory τ. For example, each attempt to balance the staff or each tic-tac-toe game is an episode. The goal of these episodes is for the agent to learn the environment and improve its strategies to maximize the cumulative reward (Ravichandiran, 2020).

Depending on which RL algorithm is used, the episodes are used differently. For example, 10 episodes at a time can run entirely independently of one another and be used to explore the environment. Then, the policies are evaluated, and the best one is varied for another 10 episodes. The goal, as mentioned before, is to find the best possible policy and do so in the most efficient way, i.e. with few iterations. Here, as already mentioned in the subsection Agent, 3.1.1, the exploration-exploitation dilemma is crucial.

RL Task Classification and Horizon Reinforcement learning tasks can be divided into episodic and non-continuous tasks. The first run iteratively in the episodes presented earlier. The agent is supposed to move from an initial state to a terminal state in this process. In the second case, no terminal state exists (Ravichandiran, 2020).

Important for episodes is the notion of the horizon; this specifies how many time steps the agent is allowed to interact with the environment until it aborts the episode or, in other words, when the lifespan of the agent ends. A distinction is made between a limited and an endless horizon. As the name implies, a predefined number of state changes is performed. It should be noted that even if the goal state is reached, the episode is not cancelled, but the action space for the goal does not change the state and the reward is zero so that there are no problems in the implementation. The same is valid for environments where the agent may be stuck in a state. An agent-environment interaction with an infinite horizon has no final state and is thus a continuous task (Ravichandiran, 2020; Bilgin, 2020).

Return and Discount Factor Another fundamental concept of RL is the return, Equation 25, of a trajectory τ. This denotes the sum of all rewards r_t over all time steps t from $t = 0$ to $t = T$. To prevent infinitely large returns, the discounted return (Equation 26) is introduced, devaluing each reward by the so-called discount factor γ^k. The value for γ is selected from the interval $[0, 1]$ and k increases with each additional reward by 1. The discounted reward favours immediate rewards, giving them more weight and devaluing those in the far future. This usage leads to finding the optimal policy since the agent should get the highest possible returns and do so in the shortest time since the longer the task runs, the less successful it is. Especially for continuous tasks this factor is important, where $T = \infty$ (Ravichandiran, 2020; Bilgin, 2020).

$$R(\tau) = r_0 + r_1 + r_2 + ... + r_T = \sum_{t=0}^{T} r_t, \tag{25}$$

$$R(\tau) = \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + ... + \gamma^n r_\infty = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{26}$$

The lower γ is, the more critical immediate rewards are, with the limit $\gamma = 0$ where all rewards after the first one are not considered. Consequently, a high factor is less punishing, and future rewards are more relevant. On the other hand, with the threshold $\gamma = 1$, all rewards are equally important, and the return can become infinitely large, as mentioned at the beginning. Therefore, this factor is crucial and must be tuned for each problem (Ravichandiran, 2020; Bilgin, 2020).

Model Models of agents in reinforcement learning are created using the Markov Decision Process (MDP). MDPs are described by the following tuple $< \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma >$ and describe memoryless, random processes, i.e. the decisions of the agent depend only on the current state and not on past states. Here, the $\mathcal{S}$ denotes the set of states the agent can observe in the environment. The transition matrix $\mathcal{P}$ describes the transition probability of all possible current states s to all following states s', which is reached by the action a, or mathematically described by the probability $P(s'|s, a)$. Furthermore, there is the reward function $\mathcal{R}$, which evaluates the reward for the transition from the states with the respective actions, mathematically $R(s, a, s')$. The last quantity describing the Markov Decision Process is the discount factor γ, which is supposed to adjust the return, as already described in the previous paragraph (Ravichandiran, 2020).

Value Function and Q Function The value function defines the value of a state (27), sometimes called the state value function, which indicates the cumulative rewards of the state under consideration for a given strategy of the agent. So in other words, the value function $V_\pi(s)$ provides the expected discounted return $R(\tau)$ for the trajectory τ from the state s, with a given policy π (Ravichandiran, 2020).

$$V_\pi(s) = \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_t = s]. \tag{27}$$

The value function can also be written recursively in the form of the so-called Bellman equation (28). The Bellman equation belongs to the most fundamental optimization equations and is central for reinforcement learning. It divides the reward of the current state into an immediate one and a discounted one, as described before. Thus, recursively all optimal solutions for smaller subsystems can be computed and transformed into a combined optimal value function $V_*(s)$ (29) (Powell, 2022).

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_\pi(s')), \tag{28}$$

$$V_*(s) = \max_\pi V_\pi(s), \tag{29}$$

$$Q_\pi(s,a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|s_t = s, a_t = a]. \tag{30}$$

The Q function should be mentioned for the completeness of the fundamental concepts of reinforcement learning. The Q function (30) extends the value function by actions. It thus has a state-action pair as a function argument. Otherwise, the idea of the value function is very similar (Ravichandiran, 2020).

3.1.3 CMA-ES

In this subsection, the policy optimization method is presented, and, in particular, the covariance matrix adaptation evolution strategy, CMA-ES, is explained. The CMA-ES is a robust optimization method with "good" convergence properties. These properties are significant for DMPs since the weights are towards the end of the trajectory. Due to the decay of the canonical system, these weights are several powers of 10 larger than at the beginning. In addition, the algorithm generally prevents convergence to local minima. Therefore, the CMA-ES is very well suited for this application of policy optimization. Moreover, as Stulp and Sigaud (2012) have described, the algorithm is significantly more effective for such optimization tasks than conventional reinforcement algorithms, such as PI^2 or REINFORCE. However, Stulp and Sigaud (2012) have pointed out that this may only be true for their application and does not have general validity.

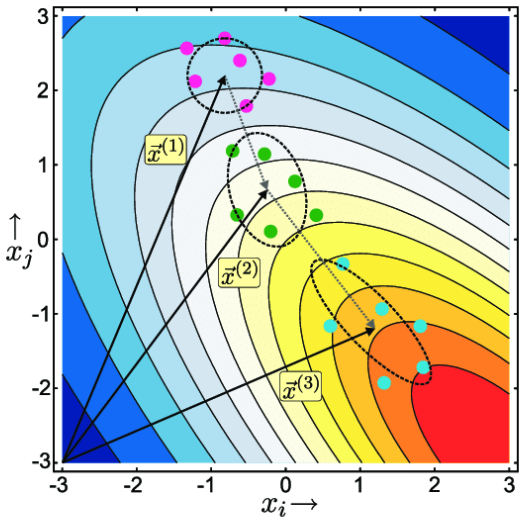


Figure 7: *This figure visualizes the CMA-ES algorithm (Shir et al., 2011).*

CMA-ES Principle The covariance matrix adaptation evolution strategy is a black-box algorithm, Algorithm 1, which is used for non-linear optimization. CMA-ES is particularly used when classical optimizers, such as gradient methods or quasi-Newton methods, do not work because of local optima, discontinuities, noise or similar problems. The algorithm uses a non-stationary, i.e. changeable, multivariate normal distribution with mean $m^{(g)}$, step-size $\sigma^{(g)}$ and covariance matrix $\boldsymbol{C}^{(g)}$, where the g indicates the generation of the algorithm. A visualization of an optimization by the CMA-ES is shown in the Figure 7 (Hansen, 2016; Shala et al., 2020).

($\mu/\mu_W, \lambda$)-CMA-ES Algorithm The CMA-ES algorithm requires several parameters, which are described in Appendix 1. Furthermore, the calculation of these parameters is also given. The setting of the parameters is done according to the proposed method of Hansen (2016).

Algorithm 1 $(\mu/\mu_W, \lambda)$-CMA-ES

set $\lambda, w_{i...\lambda}, c_\sigma, d_\sigma, c_c, c_1, c_\mu$ ▷ number of samples per iteration, at least two, generally > 4
initialize $m, \sigma, C = I, p_\sigma = 0, p_c = 0, g = 0$ ▷ initialize state variables
while *not terminate* **do**
  for i in $\{1, ..., \lambda\}$ **do** // sample λ new solutions and evaluate them
    x_i = sample_multivariate_normal(mean $= m$, covariance_matrix $= \sigma^2 C$)
    f_i = fitness(x_i)
  end for
  $x_{1,..,\lambda} \leftarrow x_{s(1),...,s(\lambda)}$ with $s(i) = \mathrm{argsort}(f_1, ..., f_\lambda)$ ▷ sort solution
  $m' = m$ ▷ for the $m - m'$ and $x_i - m'$ computation
  $m \leftarrow$ update_m$(x_1, ..., x_\lambda)$ ▷ move mean to better solution
  $p_\sigma \leftarrow$ update_ps$(p_\sigma, \sigma^{-1}C^{-1/2}(m - m'))$ ▷ update isotropic evolution path
  $p_c \leftarrow$ update_pc$(p_c, \sigma^{-1}(m - m'), \|p_\sigma\|)$ ▷ update anisotropic evolution path
  $\sigma \leftarrow$ update_sigma$(\sigma, \|p_\sigma\|)$ ▷ update step-size using isotropic path length
  $C \leftarrow$ update_C$(C, p_c, (x_1 - m')/\sigma, ..., (x_\lambda - m')/\sigma)$ ▷ update covariance matrix
end while
return m or x_1

Sampling In CMA-ES, λ samples are drawn for each generation g. Samples $\vec{x}_i^{(g)}$ are drawn from multivariate normal distribution(31) with mean $m(g)$, step-size $\sigma^{(g)}$ and covariance matrix $\boldsymbol{C}^{(g)}$. Subsequently, the drawn samples are evaluated with a fitness function, respectively cost function.Furthermore, the eigenvalue decomposition(32) of the matrix $\boldsymbol{C}^{(g)}$ is calculated so that $\boldsymbol{C}^{(g)^{-\frac{1}{2}}}$ can be computed (Hansen, 2016).

$$\begin{aligned} \mathbf{x}_i &\sim \mathcal{N}(\mathbf{m}^{(g)}, {\sigma^{(g)}}^2 \boldsymbol{C}^{(g)}), \\ &\sim \mathbf{m}^{(g)} + \sigma^{(g)} \times \mathcal{N}(0, \boldsymbol{C}^{(g)}). \end{aligned} \tag{31}$$

$$\begin{aligned} \boldsymbol{C} &= \boldsymbol{B}\boldsymbol{D}^2\boldsymbol{B}^T, \\ \boldsymbol{C}^{-\frac{1}{2}} &= \boldsymbol{B}\boldsymbol{D}^{-1}\boldsymbol{B}^T. \end{aligned} \tag{32}$$

Selection and Recombination The next step is to sort the samples according to their performance on the fitness function. From these sorted samples, the updated mean is calculated from μ best samples with the equation (33) (Hansen, 2016).

$$\{x_{i:\lambda} | i = 1...\lambda\} = \{x_i | i = 1...\lambda\} \text{ and } f(x_{1:\lambda}) \leq ... \leq f(x_{\mu:\lambda}) \leq f(x_{\mu+1:\lambda}).$$

$$\begin{aligned} \mathbf{m}^{(g+1)} &= \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}, \\ &= \mathbf{m}^{(g)} + \sum_{i=1}^{\mu} w_i(\mathbf{x}_{i:\lambda} - \mathbf{m}^{(g)}). \end{aligned} \tag{33}$$

Step-size Control Now the so-called conjugate evolution path $\vec{p_\sigma}$ is constructed, which updates the step size σ. The updates of the two quantities are given in the equations (34) and (35) (Hansen, 2016).

$$\mathbf{p}_\sigma^{(g+1)} = (1 - c_\sigma^{(g)}) + \sqrt{1 - (1 - c_\sigma^{(g)})^2 \mu_{eff}}\, \boldsymbol{C}^{(g)^{-\frac{1}{2}}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}. \tag{34}$$

$$\sigma^{(g+1)} = \sigma^{(g)} \times \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\left\|\vec{p_\sigma}^{(g+1)}\right\|}{E\|\mathcal{N}(0,I)\|} - 1\right)\right), \tag{35}$$

$$\text{with } E\|\mathcal{N}(0,I)\| = \sqrt{2}\Gamma(\frac{n+1}{2})/\Gamma(\frac{n}{2}) \approx \sqrt{n}(1 - \frac{1}{4n} + \frac{1}{21n^2}).$$

Covariance matrix adaption Finally, the covariance matrix is adapted. To accomplish the adaption, the evolution path p_c and the covariance matrix $\boldsymbol{C}$ must be recalculated, these are computed with the following equations (36), (37) and (38).

$$\mathbf{p}_c^{(g+1)} = (1 - c_c)\mathbf{p}_c^{(g)} + h_\sigma\sqrt{c_c(2 - c_c)\mu_{eff}}\ \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}. \tag{36}$$

$$\mathbf{y}_{i:\lambda}^{(g+1)} = \frac{\mathbf{x}_{i:\lambda} - \mathbf{m}^{(g)}}{\sigma^{(g)}}.$$

$$w_i^{(o)} = w_i \times \begin{cases} 1 & \text{if } w_i \geq 0, \\ \dfrac{n}{\left\|\boldsymbol{C}^{(g)^{-\frac{1}{2}}}\mathbf{y}_{i:\lambda}^{(g+1)}\right\|^2} & \text{else.} \end{cases} \tag{37}$$

$$\boldsymbol{C}^{(g+1)} = (1 + c_1\delta(h_\sigma) - c_1 - c_\mu\sum_{j=1}^{\lambda} w_j)\boldsymbol{C}^{(g)} + c_1\mathbf{p}_c^{(g+1)}\mathbf{p}_c^{(g+1)^T} + c_\mu\sum_{i=1}^{\mu} w_i^o\mathbf{y}_{i:\lambda}^{(g+1)}(\mathbf{y}_{i:\lambda}^{(g+1)})^T. \tag{38}$$

$$\text{with } \delta(h_\sigma) = (1 - h_\sigma)c_c(2 - c_c) \leq 1.$$

$$\text{with } h_\sigma = \begin{cases} 1 & \text{if } \dfrac{\left\|\mathbf{p}_\sigma^{(g+1)}\right\|}{\sqrt{1 - (1 - c_\sigma)^{2(g+1)}}} < (1.4 + \dfrac{2}{n+1})E\|\mathcal{N}(0,I)\| \\ 0 & \text{else.} \end{cases}$$

It is important to note that there are two case distinctions in the calculations, one time in the calculations of the weights and the other time it describes the Heavyside function h_σ. The case that $h_\sigma = 0$ becomes, is rare. Nevertheless this is needed if the target function is changed with the time or the step size was chosen too small with the initialization (Hansen, 2016).

3.2 Movement Primitives and Representations

One of the essential capabilities in robotics is the generation of trajectories. Over the last few decades, numerous methods have been developed to accomplish trajectory creation. This section aims to provide the mathematical foundations of dynamic movement primitives required for the framework to apply imitation and reinforcement learning. Nevertheless, first, the most important properties that trajectory generation and DMPs should possess will be discussed.

3.2.1 Properties of Trajectory Generators

The methods should create compact trajectories, which means a few parameters define the trajectories. Furthermore, they should be smooth, which means that the velocities and acceleration do not have any discontinuities and be as flexible as possible for various applications. Some elementary calculation methods are not as general and flexible as DMPs, so this thesis will not discuss them. These include manual point-to-point computation, splines, and similar techniques. In this section, we will instead focus on dynamic movement primitives. In particular, the applicability of the methods for reinforcement learning, RL, and imitation learning, IM (Ijspeert et al., 2013).

3.2.2 Properties of Dynamic Movement Primitives

Dynamic Movement Primitives, DMPs, are a method to model weakly nonlinear differential equation systems. For this purpose, a well-understood attractor model has been modified with a forcing term that produces stable, time-invariant and space scalable models, with very few variables. Attractors are stable "paths" in differential equation fields, which lead to a fixed final state of the system, which uses discrete DMPs, or cyclic sequences, which Harmonic DMPs generate. The states are not identical but similar enough to be recognized as a "pattern". This method provides a compelling way for robotics to generate trajectories with learning algorithms. At the end of this section, the application of DMPs, as mentioned before, to via-point trajectories and imitation learning will be explained for this purpose (Ijspeert et al., 2013).

3.2.3 Mathematical formulation of the Dynamic Movement Primitives

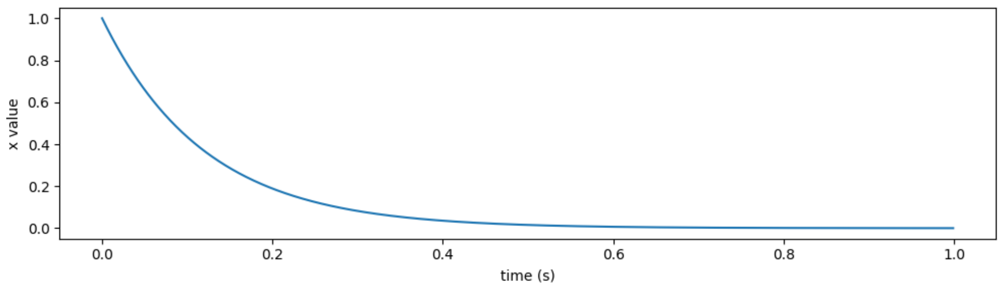
For illustration, the mathematical methods for systems with one DOF are first presented. There are two ways to write a spring-damper system using a differential equation. As a second-order equation (ref eq : DMP2) or as a first-order equation(40); these are also called transformation system. The τ is a constant used to accelerate or decelerate the motion if necessary or can be used for temporal scaling. α_z and β_z are time constants which are usually defined beforehand and indicate the damping properties of the system.

$$\tau \ddot{y} = \alpha_z(\beta_z(g - y) - \dot{y}) + f, \tag{39}$$

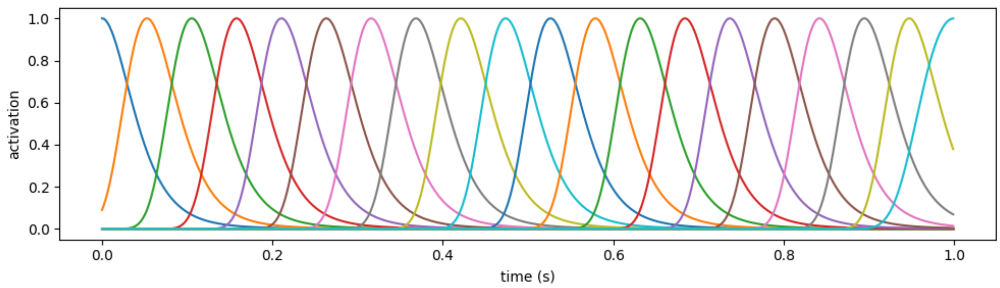
$$\begin{aligned} \tau \dot{z} &= \alpha_z(\beta_z(g - y) - z) + f, \\ \tau \dot{y} &= z. \end{aligned} \tag{40}$$

Furthermore, the system has a point attractor at its final position $(z, y) = (0, g)$. The f-function denotes a non-linear function, called forcing term, which results in a globally stable differential equation system in the case of 0. However, since f is not supposed to be zero but, as indicated before, a non-linear function, we use it for learning. Thereby its form differs depending on the application, depending on whether it is a discrete or a rhythmic motion (Ijspeert et al., 2013; Rueckert and d'Avella, 2013).

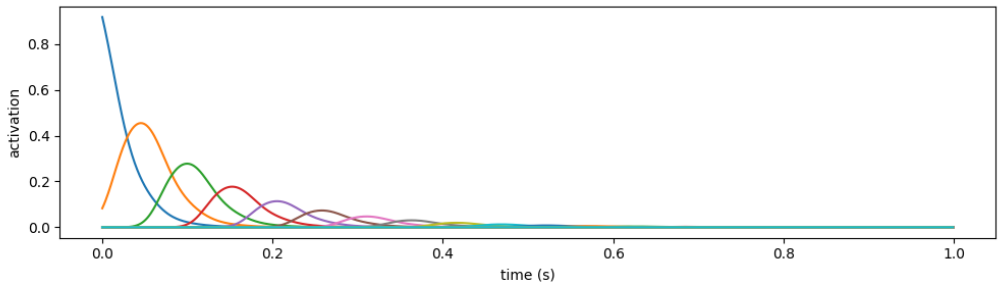
Discrete Dynamic Movement Primitives In the case of discrete dynamic motion primitives, various forcing terms f can be used, but the form described in (41) has been established for machine learning. Here f is built as a sum of N normalized weighted basis functions Ψ_i, Figure 8b, where in general the Gaussian functions (44) are used as basis functions. Whereby this notation also exists in adapted form (43), as shown in Figure 8c, with the help of a canonical system (42), Figure 8a. The α_x serves here as a constant and is chosen so that the initial state of $x_0 = 1$ and x converges monotonically to 0 at goal state, g (Ijspeert et al., 2013).



(a) *Canonical system*



(b) *Gaussian basis functions*



(c) *Gaussian basis functions scaled by the canonical system*

Figure 8: ***(a)*** *shows the exponential decay of the canonical system,* ***(b)*** *the Gaussian basis functions and* **c** *the basis function scaled by the canonical system.*

$$f(t) = \frac{\sum_{i=1}^{N} \mathbf{\Psi}_i(t) w_i}{\sum_{i=1}^{N} \mathbf{\Psi}_i(t)}, \tag{41}$$

$$\tau \dot{x} = -a_x x, \tag{42}$$

$$f(x) = \frac{\sum_{i=1}^{N} \mathbf{\Psi}_i(x) w_i}{\sum_{i=1}^{N} \mathbf{\Psi}_i(x)} x(g - y_0), \tag{43}$$

$$\mathbf{\Psi}_i(x) = \exp\left(-\frac{1}{2\sigma_i^2}(x - c_i)^2\right). \tag{44}$$

For the Gaussian basis, here c_i are the midpoints, and σ_i are the widths of each Gaussian function and are previously defined. In (43) y_0 denotes the initial state of $y(t = 0) = y_0$. Only the weights w_i are to be varied in RL or IM. Because these change the trajectory, all other constants scale only the path (Ijspeert et al., 2013).

Rhythmic Dynamic Movement Primitives The other variant of DMPs is rhythmic, which has a limit cyclic system instead of a point attractor. These systems also have to be transformed into a canonical system (45) first. An essential property of the canonical system is that if a robot has several degrees of freedom, which should be coupled, the DMPs can be joined with its help.

$$\tau\dot{\phi} = 1, \text{with } \phi \in [0, 2\pi], \tag{45}$$

$$f(x) = \frac{\sum_{i=1}^{N} \mathbf{\Psi}_i(t) w_i}{\sum_{i=1}^{N} \mathbf{\Psi}_i} r, \tag{46}$$

$$\mathbf{\Psi}_i(x) = \exp(h_i(\cos(\phi - c_i) - 1)). \tag{47}$$

The simplest variant to establish such a canonical system is a phase oscillator. Furthermore, for rhythmic dynamic motion primitives, no Gaussian basis is used anymore but a Mises basis (47). The form of the forcing term changes to the following form (46) (Ijspeert et al., 2013).

4 Framework

This chapter describes the basic structure of the CPS fameworks and its parts. First of all, the software components and the programs used are described briefly. Also the interfaces of the framework are listed and how they can be used. Furthermore, the Franka robot arm and its specifications are described. Finally, the application of the learning algorithm with dynamic movement primitives in combination with reinforcement learning and imitation learning within the CPS framework is presented.

4.1 Software

Basically, the framework was programmed with Python, since this programming language offers a relatively easy access to ROS in contrast to C++. However, ROS nodes written in Python are much slower and therefore it makes sense to write certain core tasks, for example the control ROS nodes, in C++ for faster execution. Apart from some Python libraries, for example numpy, sympy or rospy, CoppeliaSim was used as simulation program, this is presented in the following.

4.1.1 CoppeliaSim

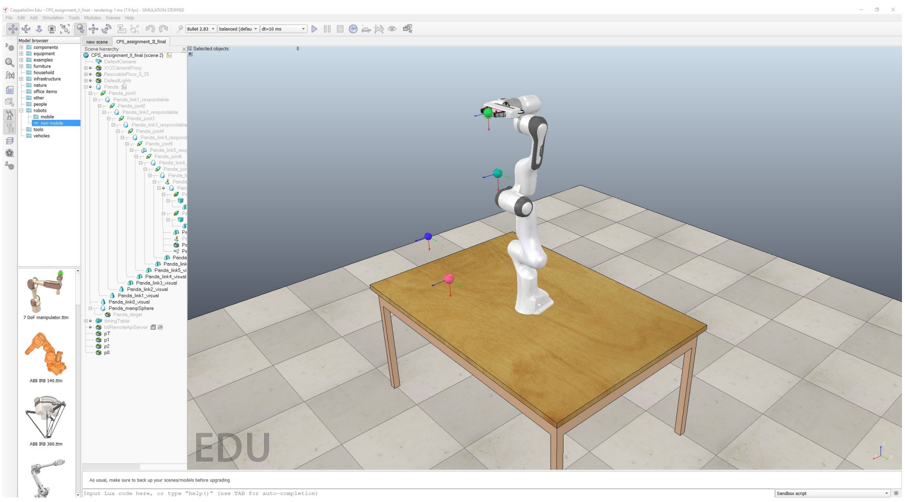


Figure 9: *This figure shows the simulation program CoppeliaSim from the company Coppelia Robotics GmbH.*

CoppeliaSim is the successor of the V-REP robot simulator from Coppelia Robotics GmbH. CoppeliaSim is a multiplatform simulator that can be communicated with via several Application Programming Interfaces (API). In this framework the interfaces ROS and ZMQ were implemented, which will be described in more detail later. Many important features are already built into the simulation program, such as multiple physics engines, collision detectors, and forward and inverse kinematics solvers. CoppeliaSim V4.3.0 was used for the framework. It is essential to use a version >=V4.3.0, because only from this version the ZeroMQ (ZMQ) interface works. Also, the embedded Python scripts will only work from this version. Within the framework, all objects in the scene, as the simulation files are called, can be created, manipulated or deleted using the ZMQ API (*CoppeliaSim* n.d.).

4.1.2 Docker

Docker is a system with which applications are packaged as so-called containers. This container contains all the necessary packages and libraries that are required for the execution of the application. Docker thus enables a platform-independent deployment of the CPS Framework as a Docker image. That means, to use the CPS Framework, only Docker is needed to run it and it works on Windows, MacOS and Linux and no additional installations are necessary (Mouat, 2015).

4.2 Interfaces

In this section, the two interfaces with which the framework communicates with CoppeliaSim and the interface with which the robot communicates, respectively, will be presented. As described before, the simulation can be controlled via the API ZeroMQ and ROS and the robot can be operated via ROS. The advantage of operating the framework via ROS is that the simulator and the real Franka arm can be operated simultaneously.

4.2.1 ZMQ

ZeroMQ enables direct control of the CoppeliaSim robot simulator. It works as an asynchronous message library and was developed for special distributed systems. The packages "pyzmq" and "cbor" are required for use in the framework. In the following the code of a few important applications is presented. First, however, the connection between the Python script and CoppeliaSim will be described (*CoppeliaSim* n.d.).

Establish ZMQ Connection As described here, establishing a connection is very easy. The default parameters for CoppeliaSim are 'localhost' with port '23000'. The 'zmqRemoteApi' script can be found by the CoppeliaSim developers on their GitHub repository.

```
# ******************************
# ZMQ Connection

from zmqRemoteApi import RemoteAPIClient

client = RemoteAPIClient('localhost',23000)
sim = client.getOnject('sim')
```

Generate Via Points in CoppeliaSim When generating new via points it starts by checking if an object with the desired name already exists, if not a dummy object is created. Then it is positioned relative to the Panda base frame.

```
if sim.getObject('/newPoint', {'noError': True}) == -1:
    sim.createDummy(0.04)
    _handle = sim.getObject('/Dummy')
    _panda_handle = sim.getObject('/Panda')
    sim.setObjectAlias(_handle, '/newPoint')
    _pose = [x,y,z]
    sim.setObjectPosition(_handle, _panda_handle, _pose)
```

Forward Kinematics in CoppeliaSim Forward kinematics can be performed in CoppeliaSim with the ZMQ API using the following code. Here, the corresponding new joint value is transmitted to each joint. This should show how easy it is to use CoppeliaSim's API. So this approach offers a good possibility of CoppeliaSim for teaching and an introduction to robot simulations with CoppeliaSim and Python.

```python
# Get the handles of each joint
joint_handles = []
robot_name='/Robot_Name'
for i in range(1, nr_Joints):
    handle = sim.getObjectHandle(robot_name + str(i))
    joint_handles.append(handle)


# get joint values of each joint
current_joint_values = []

for joint in joint_handles:
    value = sim.getJointPosition(joint)
    current_joint_values.append(value)

# set new joint values to each joint
new_joint_values = [a1, a2,..., an]

for i in range(len(joint_handles)):
    new_value = current_joint_values[i] + new_joint_values[i]
    sim.setJointPosition(joint_handles[i],newvalue)
```

4.3 Robot Operating System (ROS) and ROS2

This section provides an overview in ROS. Furthermore, the novelties of ROS2 are shown and weighed up whether an early switch pays off, since ROS is only maintained until 2025.

4.3.1 ROS basics

ROS - Robot Operating System - is a low-level software framework for robot platforms, which can be developed with C++, Python or Lisp. It consists of the so-called roscore with the communication components and the ros packages, which can be created and provided by companies, research groups or individuals. Ros packages are divided into smaller programs, so-called nodes, which can communicate with each other. These communication channels are divided into three types (*ROS-wiki* n.d.; *Generation Robots* n.d.; Joseph, 2018):

- ROS Topics are used for data streams, for example sensor data such as speed data, which can then be subscribed to by other nodes.
- ROS Service is a synchronous client/server communication between a service client and a service server. Synchronous messages are characterized by sending a request and blocking until the response comes. Therefore, they should only be used for short-lived processes, otherwise the client will be stuck for a long time. It is important that ros services are unique and defined by a name and by the data types of the message.
- ROS Action is an asynchronous communication, that means the client is not blocked waiting for a response. In this process, the client sends a destination and receives a result back at the end. While the process is executed, a feedback is sent back, which indicates the current state of the process. Furthermore, the running process can be aborted by the client at any time.

The goal of ROS is not to be the most comprehensive framework, but to give the greatest possibilities to programmers and robot developers. Furthermore, ROS is committed to the following goals (*ROS-wiki* n.d.; *Generation Robots* n.d.):

- Scalability: The programs run on small, large but also very large systems.
- Language variety: ROS packages can be developed, as before already described, with C++, Python and Lisp, furthermore still additional programming languages can communicate by means of modules or Toolboxes e.g. Matlab with ROS.
- Lightweight: ROS is built as light as simple, so that the freedom of design of the developers is as large as possible. In addition, this can be written very good reusable programs which are platform independent.
- Open Source: ROS and all its basic libraries are free and open source, making license management very simple.

4.3.2 ROS1 vs ROS2

To avoid confusion, ROS1 and ROS2 are used for the two ROS versions in the following subsection. Since ROS1 does not meet some industry requirements, including real-time capability, security and safety, and integration is very difficult or impossible without disrupting the functionality of older packages, ROS2 was developed from the mid-2010s.
Now to the innovations of ROS2. The new libraries for Python and C++ are in comparison to ROS1 much more similar whereby the readability of nodes in different programming languages is facilitated. Furthermore, the integration of additional languages has been greatly facilitated, so you can also code in Java, for example.
In contrast to ROS1 there is a clear convention in ROS2 how nodes have to be written, namely with object-oriented programming. This again increases the readability as well as the reusability of code. Furthermore, multiple nodes can be generated in one Python script because the ROS components are generated as objects. A further innovation is that Launch files must be written no longer only in XML format, but starting from ROS2 also as Python Script can be provided, whereby the configuration possibilities are greatly increased.
A large difference of ROS1 to ROS2 is that ROS2 does not have a ROS master. The reason for this is that there are no more global parameters but all parameters are associated with a node, so each node is in principle a separate server. Furthermore, ROS services are asynchronous since ROS2 and can also be equipped with callback functions. Of course they can also be changed to sychron. Actions are now part of the ROSCore and must no longer be used as a modified topic. With ROS2 the feature Quality of Service, QoS, is introduced whereby the possibility is introduced to operate ROS in networks with not good connection qualities, since messages must not always arrive.
The development environment has also changed drastically, so catkin is not used for building the packages in ROS2 but the new building system is called "ament". Furthermore, packages containing C++ and Python scripts can no longer be written as easily as in ROS1. C++ packages hardly differ from ROS1 packages, Python Scripts must be installed starting from ROS2, for which the new Python package structure was created. Nevertheless, a Python C++ package can still be created via more complex settings.
Another novelty is that ROS2 is supported by all three major operating systems, making it finally possible to create and run ROS projects with Windows and MacOS. ROS2 can also be used on embedded systems (*ROS2 Documentation* n.d.; *The Robotics Back-End* n.d.).

4.4 Franka Emika Panda

In this section, the Franka Emika robot from the company Franka Emika GmbH will be presented. The robot has seven degrees of freedom. Each joint can be controlled and has its own torque sensor, which is essential for torque control. In table 2 the Denavit-Hartenberg parameters are shown which are essential for the calculations in the framework. In APPENDIX C.2 the joint limits and the sensor specifications of the robot are given. The robot arm is equipped with a mounting flange DIN ISO 9409-1-A50 which allows to mount robot hands or other tools as long as the total load does not exceed 3kg. Also the sensor technology can be extended, in case the sensor data should be processed in the calculation process, a connection to ROS is required which can be achieved ,e.g., by means of Arduino.

Joint	a(m)	d(m)	α(rad)	θ(rad)
Joint 1	0	0.333	0	θ_1
Joint 2	0	0	$-\frac{\pi}{2}$	θ_2
Joint 3	0	0.316	$\frac{\pi}{2}$	θ_3
Joint 4	0.0825	0	$\frac{\pi}{2}$	θ_4
Joint 5	-0.0825	0.384	$-\frac{\pi}{2}$	θ_5
Joint 6	0	0	$\frac{\pi}{2}$	θ_6
Joint 7	0.088	0	$\frac{\pi}{2}$	θ_7
Flange	0	0.107	0	0

Table 2: *This table lists the Denavit-Hartenberg parameters of the Panda robot arm.*

4.5 State of the Art Learning Algorithms

In the following, the learning methods used in the framework are presented. Both Imitation Learning and Reinforcement Learning were implemented using dynamic movement primitives. For Imitation Learning, the equations for one dimension are sufficient because each dimension is considered separately. However, in reinforcement learning, one DMP must be generated and learned for each dimension, i.e., if the end effector is considered three dimensions for the position and, if necessary, three for the orientation or in joint space, one dimension for each joint. As mentioned in the previous chapter, the coupling of the systems may be necessary. However, it may become challenging because the DMPs are decoupled systems and robots, kinematic chains, are not decoupled (Rueckert and d'Avella, 2013; Schaal et al., 2003).

4.5.1 DMPs and Imitation Learning

Dynamic motion primitives provide a convenient approach to imitation learning because of their mathematical formulation. Here, trajectories presented by humans or, for example, by optical recordings should be imitated. In general, multiple demonstrations of a task are averaged, and the trajectory, velocity, and acceleration functions are substituted into the equation (48), computing the target forcing term f_{target}. For imitation learning, linear weighted regressions were proposed in Schaal et al. (2003) for this purpose. In contrast, Paraschos et al. (2018) proposed a ridge regression for PROMPs, which is used analogously for DMPs in this thesis.

$$f_{target} = \tau^2 \ddot{y}_{demo} - \alpha_z[\beta_z(g - y_{demo}) - \tau \dot{y}_{demo}], \tag{48}$$

$$f_{model} = \mathbf{\Psi w}. \tag{49}$$

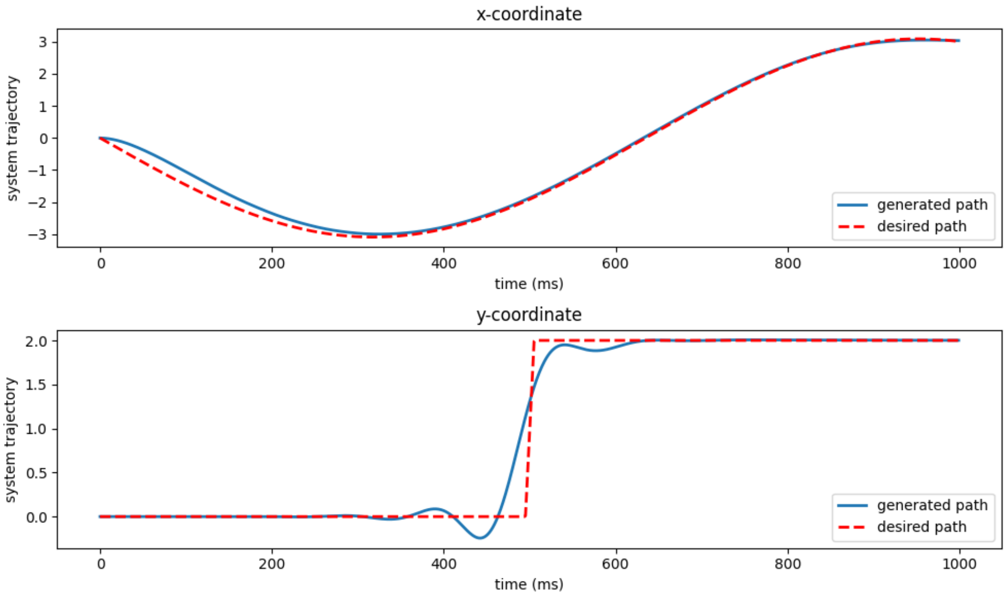


Figure 10: *In this figure an example of imitation learning is shown.*

Then, together with the model forcing term f_{model} given by (49) a cost function (50) is created. Subsequently, the weights of the model forcing term are estimated in (51). Where λ is set very small, e.g. 1e-12, because larger values degrade the estimation (Paraschos et al., 2018).

$$
\begin{aligned}
J &= \frac{1}{2}(f_{target} - f_{model})^T (f_{target} - f_{model}), \\
&= \frac{1}{2}(f_{target} - \mathbf{\Psi w})^T (f_{target} - \mathbf{\Psi w}),
\end{aligned} \tag{50}
$$

$$
w_i = (\mathbf{\Psi}^T \mathbf{\Psi} + \lambda \mathbf{I})^{-1} \mathbf{\Psi} f_{target}, \tag{51}
$$

$$
w_i = (\mathbf{\Psi}^T \mathbf{\Psi} + \lambda \mathbf{\Gamma}^T \mathbf{\Gamma})^{-1} \mathbf{\Psi} f_{target}. \tag{52}
$$

Furthermore, in Paraschos et al. (2018) an adapted version of (51) was presented, which additionally minimizes the jerk of the trajectory (52). For this purpose, the third derivatives, Γ, of Ψ are calculated, and $\Gamma^T\Gamma$ is used instead of the unit matrix I in the ridge regression.

4.5.2 DMPs and Reinforcement Learning

The following subsection presents a method for learning dynamics motion primitives with via-points. This method uses CMA-ES, a policy search method presented in the previous section, which evaluates and optimizes the policy after each episode. The method was adopted from Rueckert and d'Avella (2013) and will now be explained.

$$
\mathbf{u}_t = \text{diag}(\mathbf{k}_{pos})(\mathbf{y}_t^* - \mathbf{y}_t) + \text{diag}(\mathbf{k}_{vel})(\dot{\mathbf{y}}_t^* - \dot{\mathbf{y}}_t). \tag{53}
$$

First, a simple p controller is set up for each DOF, where y_t^* and $\dot{y}_t^*$ are the desired trajectory obtained by integrating the equation (39) and y_t and $\dot{y}_t$ were simulated. This controller can also be described in vector notation as in (53). With this, the simulation is performed. If the energy consumption of the controller is to be minimized, the function values are summed up or saved to be used for the evaluation from the objective function of the CMA-ES (Rueckert and d'Avella, 2013).

$$\begin{aligned} C(\tau) =&(\mathbf{g}-\mathbf{x}_t)^T\,\mathbf{R}_{pos}\,(\mathbf{g}-\mathbf{x}_t)\,+ \\ &\sum_{i=1}^{N}(\mathbf{g}_i-\mathbf{x}_{t=t_i})^T\,\mathbf{R}_{via}\,(\mathbf{g}_i-\mathbf{x}_{t=t_i})\,+ \\ &(\dot{\mathbf{g}}-\dot{\mathbf{x}}_\mathbf{t})^T\,\mathbf{R}_{vel}\,(\dot{\mathbf{g}}-\dot{\mathbf{x}}_\mathbf{t})\,+ \\ &\sum_{t=0}^{t_{fin}}u_t^T\,\mathbf{H}_E\,u_t. \end{aligned} \tag{54}$$

The next step is to define an objective function, or cost function, for the CMA-ES. There is an uncountable possibility of how this can be constructed. One of these variants is presented in equation (54), which divides into four parts. The first part penalizes not reaching the target position, and the second punishes not passing the N via points with position $\mathbf{g}_i$ at times t_i. The third forces the policy to a final velocity $\dot{\mathbf{g}}$ and the last part penalizes too high energy consumption of the controller. Here, the matrices $\mathbf{R}_{pos}$, $\mathbf{R}_{via}$, $\mathbf{R}_{vel}$ and $\mathbf{H}_E$ give the cost of each error. It is important to note that the cost function can be adjusted. Individual parts can be omitted, or the final velocity is often set to zero, reducing the third part to $\dot{\mathbf{x}}_\mathbf{t}{}^T\mathbf{R}_{vel}\,\dot{\mathbf{x}}_\mathbf{t}$. In addition, for example, the running time can also be penalized. The choice of the previously mentioned matrices is crucial in this case (Rueckert and d'Avella, 2013).

Procedure This state-of-the-art reinforcement learning algorithm was first presented in Rueckert and d'Avella (2013). The process of the RL algorithm, which is shown in Figure 11, starts with the design of the objective function, as described in 4.5.2, of the policy optimizer - followed by the initialization of the weights of the DMPs. Here, the number of dimensions, i.e., whether the trajectory is to be learned in task or joint space, and the number of basis functions must be determined. For movements in the task space, it is further possible to decide whether only the position or the orientation of the end effector should be considered. Furthermore, it should be noted that trajectories in Joint Space are much smoother because the inverse kinematics does not have to be solved by the controller. Therefore no discontinuities or singularities can occur. The initialization concludes with the definition of the start, goal and waypoints, and the time scaling τ.

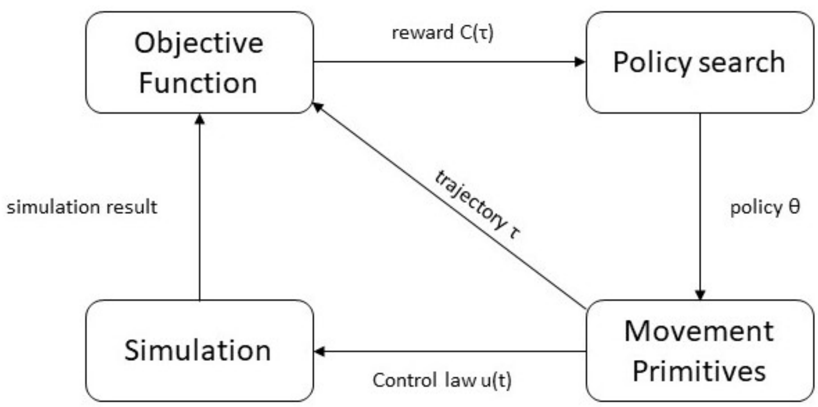


Figure 11: *This figure shows the procedure of reinforcement learning algorithm in combination with CMA-ES.*

Subsequently, the reinforcement learning algorithm starts with the first run of the policy search using CMA-ES. As described in 3.1.3, λ samples are drawn, and subsequently, the DMPs are rolled out. Then, depending on whether the trajectory is optimized for energy, a simulation of the trajectory must be performed, and the control law $\mathbf{u}_t$ is computed. Afterwards, the DMPs and, if necessary, the results of the simulations are evaluated with the objective functions. As the last step, all λ evaluations of the cost function in the CMA-ES are used to adapt the mean, covariance matrix and step size of the samples. After that, the algorithm starts again, and more samples with the new means and covariances are drawn. The algorithm ends once the step size or the best evaluation of the objective function becomes small enough or a maximum number of iterations has been performed. Note that by eliminating the energy optimization, the computation time of the algorithm is drastically reduced.

In this work, two variants of via-point trajectory generation were performed. On the one hand, the trajectories were wholly described in the task space, and an inverse controller needs to be used. The other method transforms the start, goal and waypoints into the joint space and calculates the trajectory of the joint angles.

5 Experiments

In this chapter, the experiments of the motor learning framework and their results are described. For this purpose, an adapted method of the previously presented reinforcement learning algorithm is presented first. Then, the experiments for testing the new variant are proposed, and their results are shown and discussed. Subsequently, an imitation learning experiment is presented. Finally, the use of the motor learning farm framework for the use cases is discussed.

5.1 Learn Via Point Movement with RL

In the course of conducting the experiments, an adapted procedure to the method presented in Chapter 3 was developed. This method proposes a scaling of the DMP weights with the increasing progression of the canonical system. For this, the weight vector of each DMP was scaled by the vector c_{scale} using the Hadamard product, as shown in (55), before rolling it out. Here, the vector c_{scale} has a logarithmic increasing appearance from 1 to $0.8a_x$. This idea compensates for the decay of the canonical system, which leads to small weights, about $\pm10^2$, near the initial state and to huge weights, about $\pm10^5$, at the end of the trajectory. The problem with the non-scaled weights here is that the DMPs combined with CMA-ES have issues with convergence and, in some cases, never converge to the desired movement trajectories.

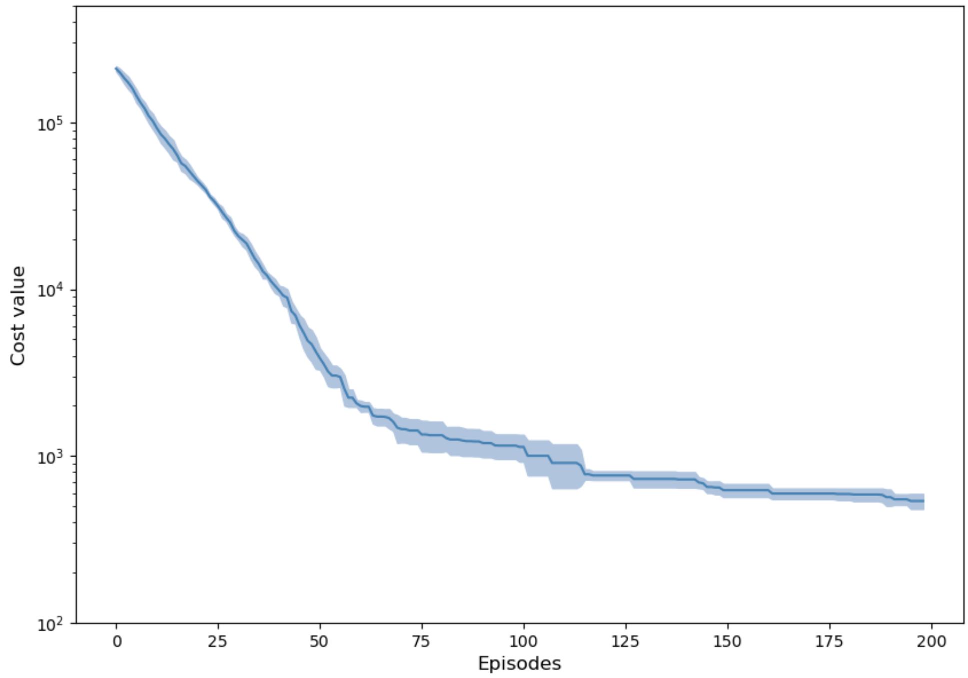
$$\begin{aligned} \mathbf{w}_{dmp} &= \mathbf{x_i} \circ \mathbf{c}_{scale}, \\ \text{with } \mathbf{c}_{scale} &= [1, \ldots, 0.8a_x]^T. \end{aligned} \tag{55}$$

For verification purposes, 5 trajectories were learned using the new and old methods. In the figures, the results for the cost function, Figures 12a and 13a, and the trajectories of the position, Figures 12b and 13b, are shown. Furthermore, it should be noted that there are no differences between the calculation times, as they require, on average, 3 minutes for 200, 8 minutes for 500 and 40 minutes for 2500 iterations. It is also clear that the time scales linearly so that parallel computing can accelerate the calculations.

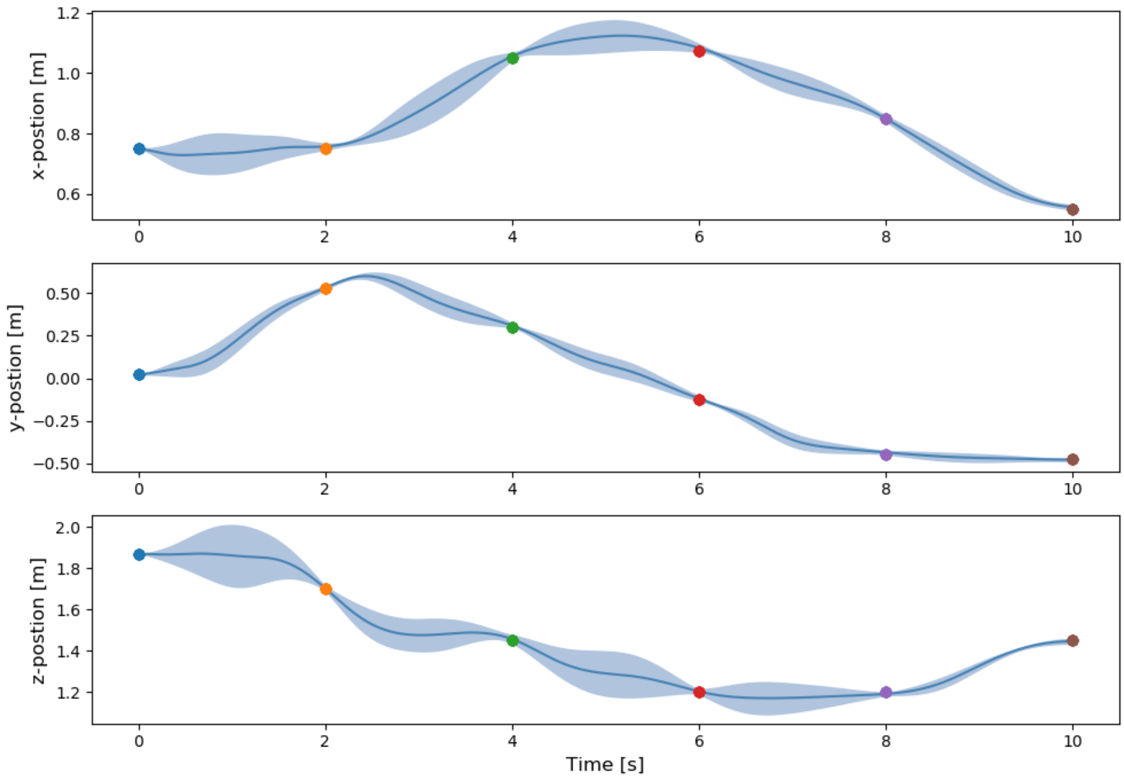
Unfortunately, the assumed effects could not be achieved, and the adapted method works as well as the non-adapted one. Moreover, the results were distorted due to an outdated version of the NumPy method. More precisely, the least square linear equation solver was changed, causing a flag mistakenly not to be set. As a result, small singular values of the matrix were not calculated correctly but set directly to zero, causing the convergence properties of the CMA-ES to no longer work correctly. As a comparison, the results of a trajectory learned with 200 iterations are given for both the scaled and non-scaled methods. Here it is evident that no improvement can be achieved. Therefore, the algorithm presented in Rueckert and d'Avella (2013) is used for the remaining computations.

	start	1	2	3	4	goal
x (m)	0.75	0.78	1.12	1.15	0.90	0.55
y (m)	0.02	0.60	0.37	-0.15	-0.53	-0.57
z (m)	1.87	1.70	1.45	1.20	1.20	1.45
t (s)	0	2	4	6	8	10

Table 3: *This table lists the start, goal and via-points of the learned trajectory.*

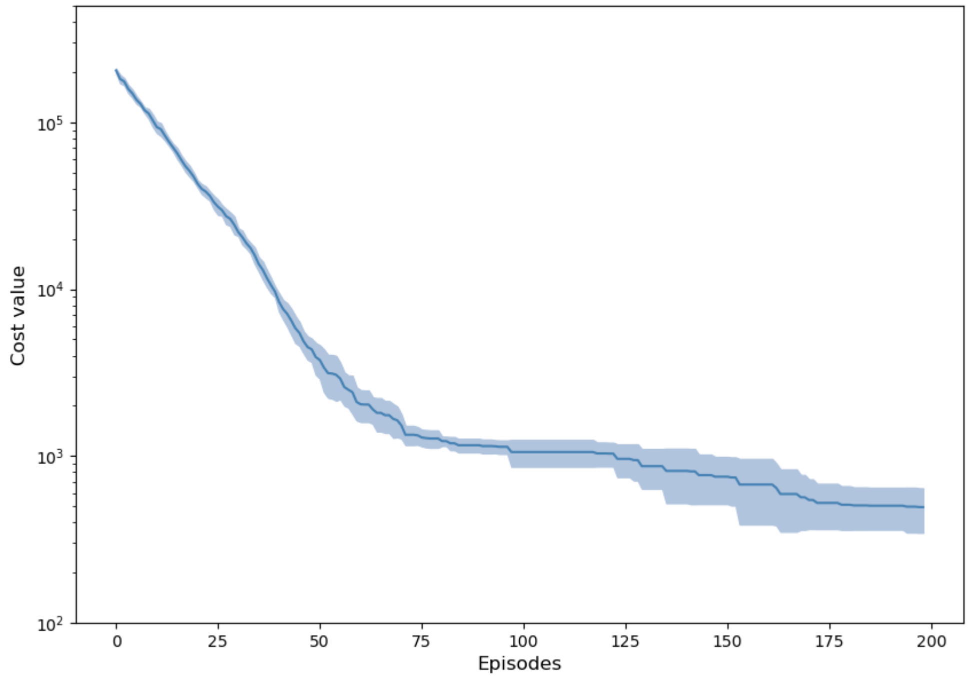


(a) *Cost Function 200 scaled*

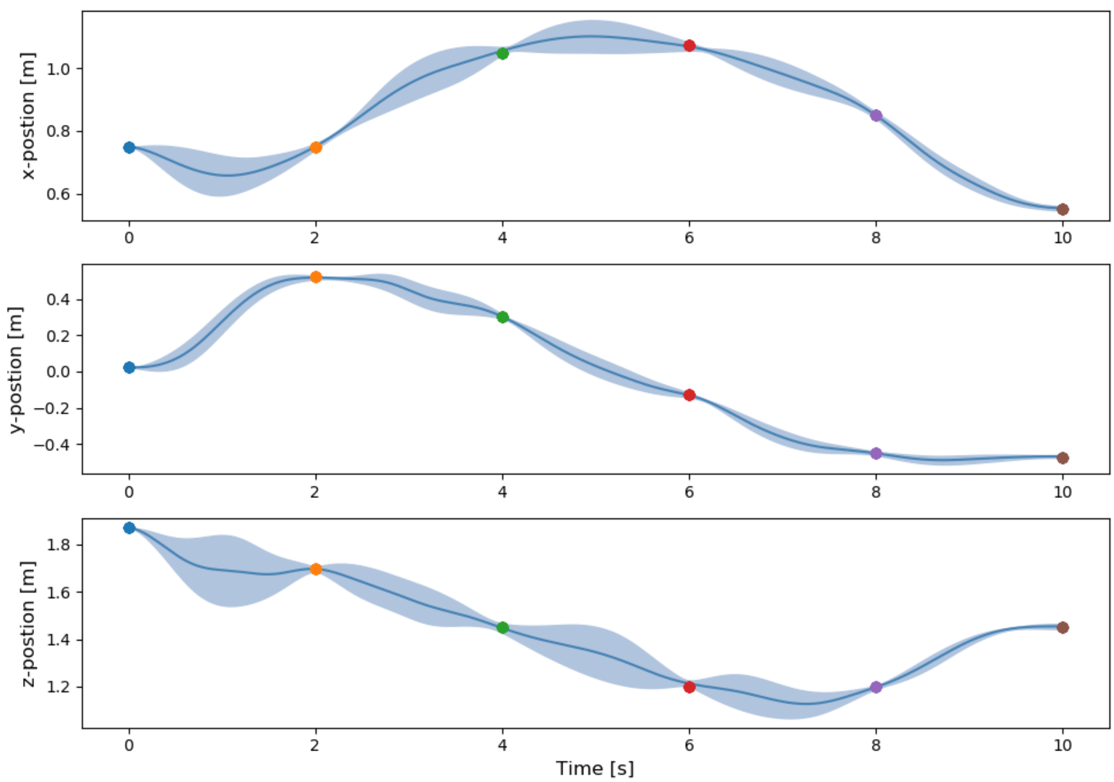


(b) *DMPs 200 scaled*

Figure 12: *The images show both the cost functions and the trajectories learned using the adapted method for 200 iterations.*



(a) *Cost Function 200 scaled*



(b) *DMPs 200 scaled*

Figure 13: *The images show both the cost functions and the trajectories learned using the adapted method for 200 iterations.*

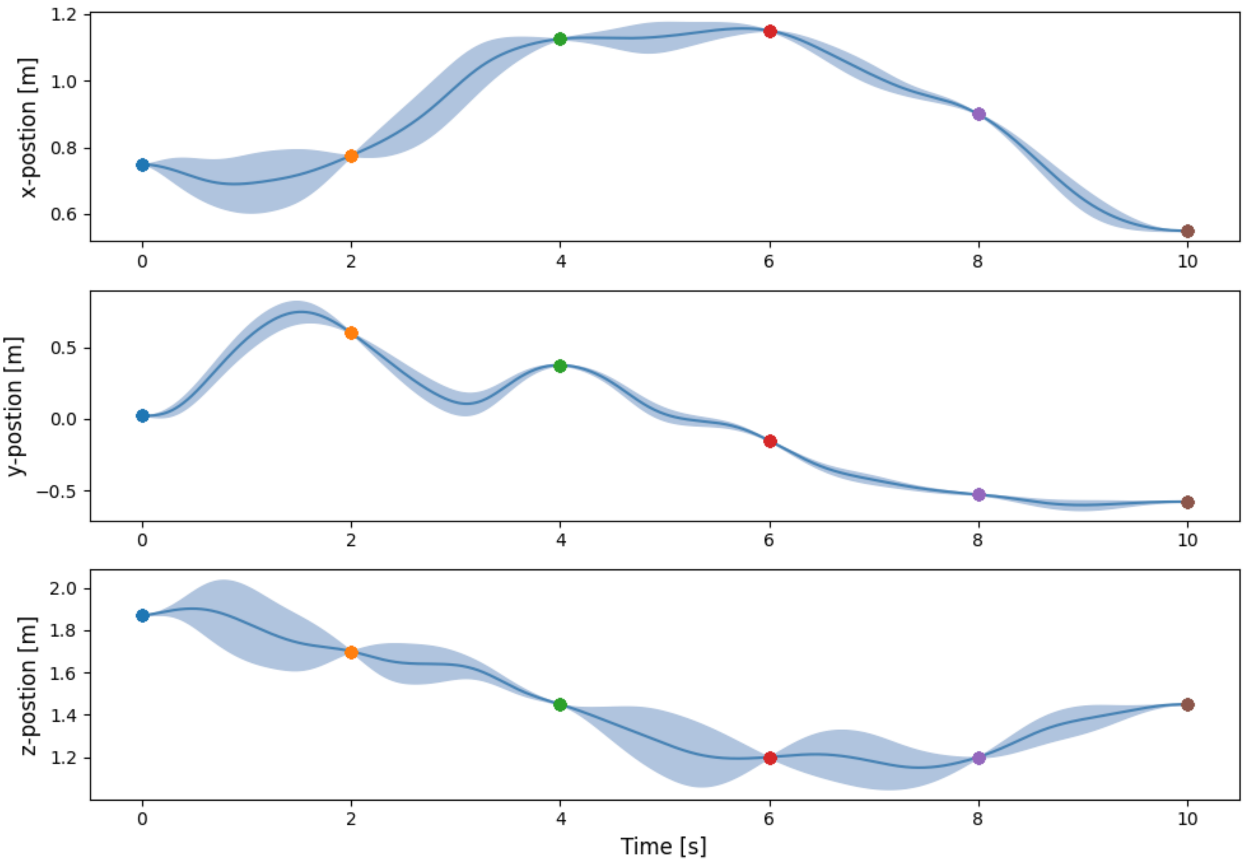
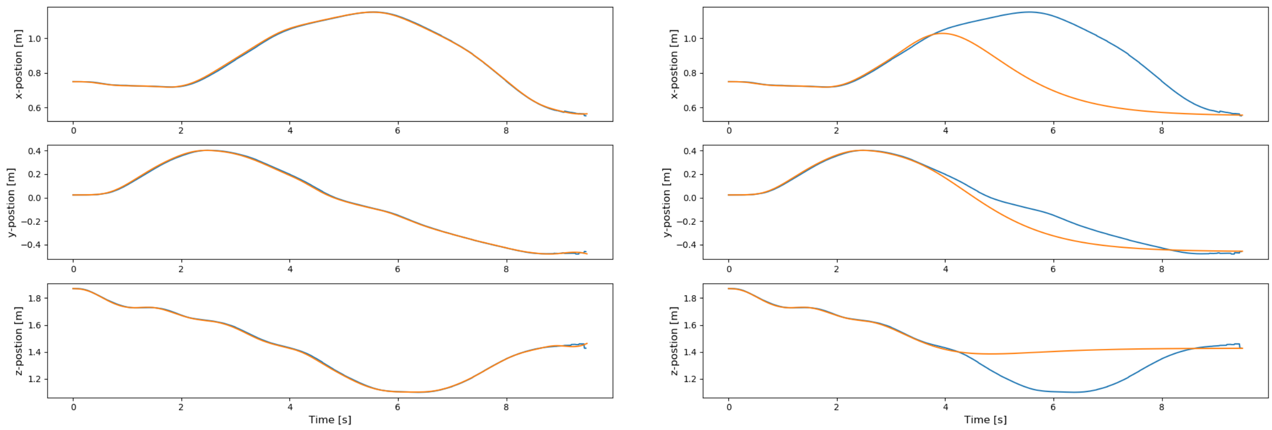


Figure 14: *The images shows the trajectories in task space learned for 2500 iterations.*

As a first application of the motor learning framework, trajectories with 4 via points were generated in the task space. In each case, 5 trajectories with 500 iterations were learned and subsequently averaged to generate reproducible trajectories. For this purpose, 3 DMPs, one for each dimension in Cartesian space, each with 25 Gaussian basis functions. The τ was set to 10 seconds. The initial position of the Franka robot was chosen as the start point, and all waypoints, as well as the goal point, are described in Table 3. For the experiment, the penalties for the target, as well as on the waypoints, were set to 10^5, the final velocity is supposed to be 0, and the penalty is 10^3. Finally, a penalty on excessive acceleration was set, which was implemented similarly to the control penalty; this is 10^{-2} for each element. These values were also used for the comparisons between scaled and non-scaled methods. The results of this experiment can be seen in Figure 14.

5.2 Imitate Robotic Movements with Dynamic Movement Primitives

For the application of imitation learning, 20 trajectories each were generated using the RL method. The four intermediate points were varied with a Gaussian distribution of $N(0, 0.05^2)$. All these 20 trajectories were checked to see if they were entirely within the workspace, which excluded 8 movements. With the rest, simulations were performed, and the end-effector's position was recorded.



(a) *Imitation Learning with Ridge Regression*

(b) *jerk optimized imitation learning*

Figure 15: *These figures show the results of imitation learning.*

Afterwards, the recordings were averaged, and various imitation learning methods were performed. The figures ABB, the results for the standard ridge regression and the adapted regression with jerk optimization are shown. The imitated regression is almost perfect. The trajectory is not completely imitated in the second method but flattened from halfway. This flattening could be due to the non-optimized hyperparameter λ.

5.3 Uses Cases

At the beginning of this thesis, three use cases for the motor framework were defined. Unfortunately, no experiments could be performed on the real robot due to hardware problems with the robot's control unit. These problems could only be solved at the end of the development period of this thesis. Therefore, only the experiments are described for the use cases "Research" and "Industry", which will be carried out afterwards.

5.3.1 Teaching

The first use case, "Teaching", has been completely processed. First, the task of the hypothetical course described in 1.4.1 is described in more detail. Then, CoppeliaSim was used as a simulation program for this application, and the communication was done with the message library ZMQ. For this purpose, a complete robot model was implemented in Python, starting with programming the transformations for the Denavit-Hartenberg parameters and the forward kinematics. Subsequently, a Jacobian Inverse Controller was developed, which, however, is not used due to singularity issues. Instead, the inverse controller of CoppeliaSim has been applied. In Figure 16, the three-dimensional end-effector motion is shown. This trajectory was calculated using the RL algorithm as a task space problem.

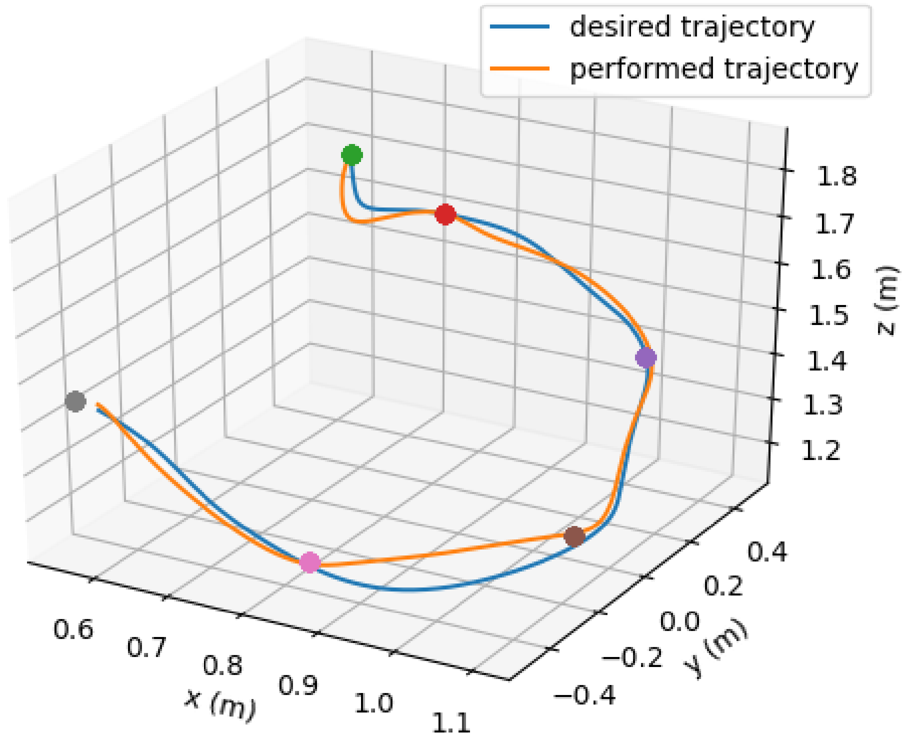


Figure 16: *This figure displays the trajectories of the desired path and the path performed by the CoppeliaSim inverse controller.*

The simulation experiments with ROS could not be fully completed. However, these will be completed, and the experiments following the master thesis will therefore not be discussed in further detail.

5.3.2 Industry

In the "Industry" use case, the focus will be on applying imitation learning. For this purpose, several movements are to be demonstrated by the robot operator. Conceivable demonstrations would be, for example, a simple industrial application, such as a pick and place process. A ROS Publisher is required, which records the joint angles. Subsequently, the data is averaged for each time and learned as an imitation using one of the two implemented regression analyses.

5.3.3 Research

The definition of experiments for the use case "research" is relatively complex since, in the end, only the simultaneous application of actual and simulated robots is to be tested. Nevertheless, the experiments from the "Teaching" and "Research" use cases can be used. Subsequently, the research of the Chair of Cyber-Physical Systems can be supported directly, and experiments in the field of motor control of serial robots can be immediately performed.

6 Discussion

This chapter reviews the results of the thesis. The methods are evaluated, and possible further work is discussed. The core task of the motor learning framework, the trajectory generation, could be implemented successfully. Both methods, the reinforcement learning algorithm and the imitation learning, work in the task space.

Reinforcement Learning However, only the trajecotry of the position and not the orientation of the end-effector were considered. The motion in task space is implemented well, but the movement is still unstable, due for actual tasks since no orientation of the end effector can be determined. For this reason, the generated movements of the end-effector are not entirely free of oscillations. Therefore, trajectories with fewer iterations were generated using the reinforcement learning algorithm and then averaged to obtain better results. As a result, the averaged trajectories have fewer oscillations, and thus the motion of the simulated robot is much smoother than before..

Imitation Learning In imitation learning, the benefit of jerk optimization could not be established because the trajectories used do not have any particular problems with jerk. Therefore the much higher computational cost is not worth it for the experiments performed. In addition, the trajectories are guided to the goal value at an early stage and thus do not perform the intended movement. However, ridge regression calculation produces desired results with very brief calculation times. In particular, the method should be further tested with the real robot.

Framework In conclusion, it is questionable whether the use of CoppeliaSim is helpful in the context of the Motor Learning Framework. CoppeliaSim is currently only used to visualise waypoints and for low-level communication via ZMQ. When using ROS, the software libraries of Franka Emika are used, which use Gazebo as simulation. Therefore, in the application of ROS, CoppeliaSim is only used to visualise the robot and the waypoints.

6.1 Conclusion

In this thesis, a motor control learning framework was developed. This framework should facilitate serial robots, especially the Panda robot arm of the company Franka Emika GmbH. For the generation of motion trajectories, two different approaches were chosen. Here, movements were modelled by Dynamic Movement Primitives and their weights were calculated using two learning methods. The reinforcement algorithm uses a policy optimizer, which uses the CMA-ES. The optimizer iteratively adjusts the weights of the DMPs based on a cost function. Furthermore, regression models for imitating motion demonstrations can be used.

Use Cases Additionally, three use cases, called "teaching", "industry", and "research", were defined for which the framework can be applied. However, due to hardware issues with the control unit of the real Panda robot, only the use case "teaching" could be considered, and the other two will be reviewed after the master thesis has been completed.

Adapted Reinforcement Learning In the course of the conducted waypoint experiment, an adapted version of the method of Rueckert and d'Avella (2013) was developed and evaluated. The weights of the dynamic movement primitives were scaled over the runtime of the canonical system to compensate for the decay. The idea of the adapted method was to achieve better and faster convergence properties for the DMPs. Unfortunately, the desired effects could not be proven because even after only 200 iterations, the performance of both methods was the same, thus no improvement could be achieved. The original assumption was supported by using an outdated Numpy function, however, the performance benefits could be eroded by using the newer version.

Reinforcement Learning Nonetheless, waypoint experiments were conducted in Task Space. The experiments demonstrated that the generation by the proposed reinforcement learning algorithm could create smooth and nearly oscillation-free motions. Furthermore, the use is straightforward. Only the waypoints have to be defined and the temporal scaling. All other parameters can be taken from the appendix and have to be adapted only marginally to the experiment.

Imitation Learning Finally, an experiment on the method of imitation learning was performed. First, trajectories were generated with the reinforcement learning algorithm, in which the positions of the waypoints varied slightly in each case and performed in a simulation. Then, the joint angle data were recorded, and two different regression models were used to computed trajectory models from the demonstrations. For the regression, classical ridge regression was used to optimize the jerk. Therefore, the third derivatives of the basis functions were utilized instead of the identity matrix.

6.2 Future Work

As further work, the evaluation of the two remaining use cases, "Industry" and "Research", is planned. In particular, the application of linking actual and simulated robots is not an easy task due to the reality gap.

Motion Capturing Therefore, an extension of the Motor Control Learning Framework by an interface to the OptiTrack system, a camera-based motion capturing system used at the Chair of Cyber-Physical Systems. This extension could make it more convenient to demonstrate trajectories using optical markers of the motion capturing system. Thus, the movements of an arm can be recorded directly and imitaiton learning can be applied to real applications, for example the use of tools.

Paralell Computing and ProMP Furthermore, optimizing the reinforcement algorithm concerning parallel computing would be beneficial, reducing the computation times. Also, the Dynamic Movement Primitives should be evaluated, so the convergence properties of the RL algorithm could be further improved with Probabilistic Movement Primitives (ProMPs). In addition, the use of ProMPs allows the application of near-real-time systems and the use of planning algorithms.

Application to other robots In addition, for the application of robots in actual experiments, the possibility of using grippers or robot hands is necessary. For this purpose, the motor control learning framework should also be extended. In addition, the application to the different robots of the Cyber-Physical Systems chair will be performed to demonstrate the universal usability of the framework. Consequently, suitable parameters for the respective robot can be determined and the motor learning framework will be further improved.

Force Control Finally, force control is to be added to the framework. Thereby, the application of the robots for collaborative use should be improved. Movement primitives can also describe force control trajectories, but the implementation is more complicated than controlling by joint angle or velocity. Nevertheless, it enables even broader use of the robot and many more applications that are not possible through conventional controllers.

Bibliography

Arulkumaran, Kai et al. (2017). "Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Processing Magazine* 34.6, pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

Bilgin, E. (2020). *Mastering Reinforcement Learning with Python: Build Next-generation, Self-learning Models Using Reinforcement Learning Techniques and Best Practices*. Packt Publishing. ISBN: 9781838644147. URL: https://books.google.at/books?id=_g8FzgEACAAJ.

Bousmalis, Konstantinos and Sergey Levine (2017). *Closing the simulation-to-reality gap for deep robotic learning*. URL: https://ai.googleblog.com/2017/10/closing-simulation-to-reality-gap-for.html.

Chou, Po-Wei, Daniel Maturana, and Sebastian Scherer (n.d.). "Improving Stochastic Policy Gradients in Continuous Control with Deep Reinforcement Learning using the Beta Distribution". In: ().

Constans, Eric and Karl B Dyer (2018). *Introduction to mechanism design: With computer applications*. CRC Press.

CoppeliaSim (n.d.). URL: https://www.coppeliarobotics.com/ (visited on 04/20/2022).

Craig, J.J. (2021). *Introduction to Robotics, eBook, Global Edition*. Pearson Education. ISBN: 9781292164953. URL: https://books.google.at/books?id=Bjw1EAAAQBAJ.

Dudek, Gregory and Michael Jenkin (2010). *Computational Principles of Mobile Robotics*. 2nd. USA: Cambridge University Press. ISBN: 0521692121.

Franka-Emika-GmbH (2018). *Panda's Instruction Handbook*.

Generation Robots (n.d.). URL: https://www.generationrobots.com/blog/de/ros-robot-operating-system/ (visited on 02/03/2022).

Gentle, James E. (2017). *Matrix Algebra: Theory, Computations, and Applications in Statistics*. 2nd. Springer. ISBN: 9783319648668.

Hansen, N. and A. Ostermeier (2001). "Completely derandomized self-adaptation in evolution strategies". In: *Evolutionary Computation* 9.2, pp. 159–195.

Hansen, Nikolaus (2016). "The CMA evolution strategy: A tutorial". In: *arXiv preprint arXiv:1604.00772*.

Ijspeert, Auke Jan et al. (2013). "Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors." In: *Neural Comput.* 25.2, pp. 328–373. URL: http://dblp.uni-trier.de/db/journals/neco/neco25.html#IjspeertNHPS13.

Joseph, Lentin (2018). *Learning Robotics Using Python: Design, Simulate, Program, and Prototype an Autonomous Mobile Robot Using ROS, OpenCV, PCL, and Python, 2nd Edition*. 2nd. Packt Publishing. ISBN: 1788623312.

Kelly, R., V.S. Davila, and J.A.L. Perez (2006). *Control of Robot Manipulators in Joint Space*. Advanced Textbooks in Control and Signal Processing. Springer London. ISBN: 9781852339999. URL: https://books.google.at/books?id=KsBRJQEeEPcC.

Koos, Sylvain, Jean-Baptiste Mouret, and Stéphane Doncieux (2013). "The Transferability Approach: Crossing the Reality Gap in Evolutionary Robotics". In: *IEEE Transactions on Evolutionary Computation* 17.1, pp. 122–145. DOI: 10.1109/TEVC.2012.2185849.

Lapan, Maxim (2020). *Deep reinforcement learning hands-on*. Packt publishing.

Lonza, Andrea (2019). *Reinforcement Learning Algorithms with Python: Learn, understand, and develop smart algorithms for addressing AI challenges*. Packt Publishing Ltd.

Lynch, Kevin M. and Frank C. Park (2017). *Modern Robotics: Mechanics, Planning, and Control*. 1st. USA: Cambridge University Press. ISBN: 1107156300.

Mahnken, Rolf (2011). *Lehrbuch der technischen Mechanik-Dynamik: eine anschauliche Einführung*. Springer-Verlag.

Mareczek, Joerg (Jan. 2020). *Grundlagen der Roboter-Manipulatoren – Band 1: Modellbildung von Kinematik und Dynamik*. ISBN: 978-3-662-52758-0. DOI: 10.1007/978-3-662-52759-7.

Mouat, A. (2015). *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media. ISBN: 9781491915929. URL: https://books.google.at/books?id=wpYpCwAAQBAJ.

Mouret, Jean-Baptiste and Konstantinos Chatzilygeroudis (2017). “20 Years of Reality Gap: A Few Thoughts about Simulators in Evolutionary Robotics”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO ’17. Berlin, Germany: Association for Computing Machinery, 1121–1124. ISBN: 9781450349390. DOI: 10.1145/3067695.3082052. URL: https://doi.org/10.1145/3067695.3082052.

Niku, Saeed B (2020). *Introduction to robotics: analysis, control, applications*. John Wiley & Sons.

Paraschos, Alexandros et al. (2018). “Using probabilistic movement primitives in robotics”. In: *Autonomous Robots* 42.3, pp. 529–551.

Powell, Warren B. (2022). *Reinforcement Learning and Stochastic Optimization: A Unified Framework for Sequential Decisions*. John Wiley and Sons Inc. ISBN: 978-1-119-81506-8.

Ravichandiran, S. (2020). *Deep Reinforcement Learning with Python - Second Edition*. Expert insight. Packt Publishing. ISBN: 9781839210686. URL: https://books.google.at/books?id=KOHrzQEACAAJ.

Ravishankar, NR and MV Vijayakumar (2017). “Reinforcement learning algorithms: survey and classification”. In: *Indian J. Sci. Technol* 10.1, pp. 1–8.

ROS-wiki (n.d.). URL: https://wiki.ros.org/en/ROS/ (visited on 02/03/2022).

ROS2 Documentation (n.d.). URL: https://docs.ros.org/en/foxy/index.html (visited on 04/03/2022).

Rueckert, Elmar and Andrea d’Avella (Oct. 17, 2013). “Learned parametrized dynamic movement primitives with shared synergies for controlling robotic and musculoskeletal systems”. In: *Frontiers in Computational Neuroscience* 7.138. DOI: 10.3389/fncom.2013.00138. URL: https://cps.unileoben.ac.at/wp/Frontiers2013bRueckert.pdf. published.

Schaal, Stefan et al. (2003). “Control, planning, learning, and imitation with dynamic movement primitives”. In: *Workshop on Bilateral Paradigms on Humans and Humanoids: IEEE International Conference on Intelligent Robots and Systems (IROS 2003)*, pp. 1–21.

Shala, Gresa et al. (2020). “Learning step-size adaptation in CMA-ES”. In: pp. 691–706.

Shir, Ofer et al. (Dec. 2011). “Evolutionary Hessian Learning: Forced Optimal Covariance Adaptive Learning (FOCAL)”. In.

Siciliano, Bruno et al. (2008). *Robotics: Modelling, Planning and Control*. 1st. Springer Publishing Company, Incorporated. ISBN: 1846286417.

Stulp, Freek and Olivier Sigaud (Oct. 2012). “Policy Improvement Methods: Between Black-Box Optimization and Episodic Reinforcement Learning”. In.

Teixeira Silva, Murilo et al. (Aug. 2017). “Alternative Analytical Solution for Position and Orientation in Electromagnetic Motion Tracking Systems”. In: *WSEAS Transactions on Systems* 16, pp. 225–233.

The Robotics Back-End (n.d.). URL: https://roboticsbackend.com/ros1-vs-ros2-practical-overview/ (visited on 04/03/2022).

Vierhaus, Ingmar et al. (2017). “Using white-box nonlinear optimization methods in system dynamics policy improvement”. In: *System Dynamics Review* 33.2, pp. 138–168. DOI: https://doi.org/10.1002/sdr.1583. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sdr.1583. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sdr.1583.

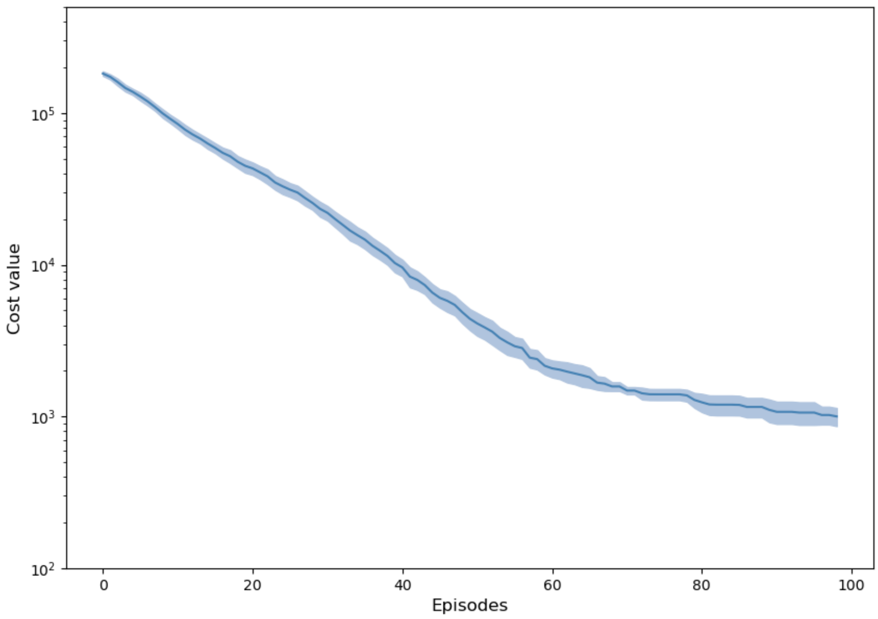
Yang, Zhuo et al. (Aug. 2017). “Investigating Grey-Box Modeling for Predictive Analytics in Smart Manufacturing”. In: V02BT03A024. DOI: 10.1115/DETC2017-67794.

Žlajpah, Leon (2008). “Simulation in robotics”. In: *Mathematics and Computers in Simulation* 79.4, pp. 879–897.

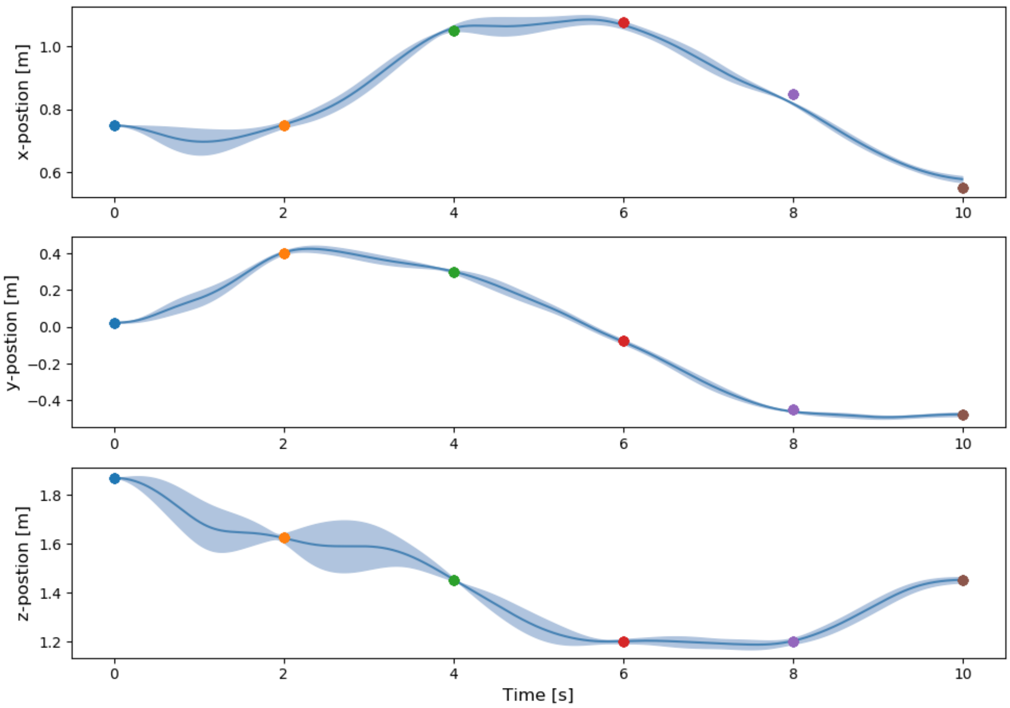
A APPENDIX ONE

A.1 Additional Results

In this section, further results of the trajectory calculation are given. The maximum number of iterations, 100 and 500, was chosen. Furthermore, a 3D trajectory calculated by imitation learning is shown, and the 3D trajectory generated by the Reinforcement Learning algorithm with 100 iterations.

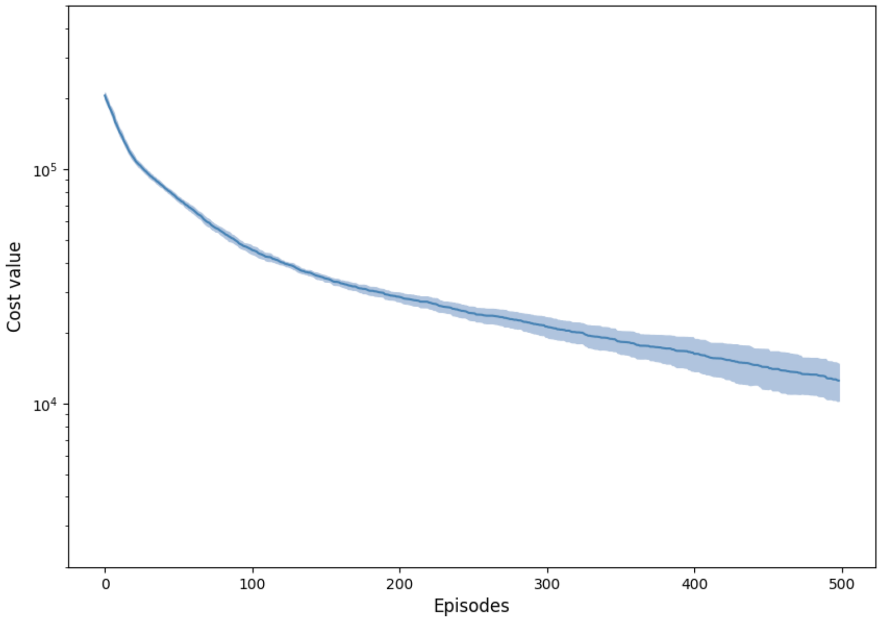


(a) *Cost Function 100*

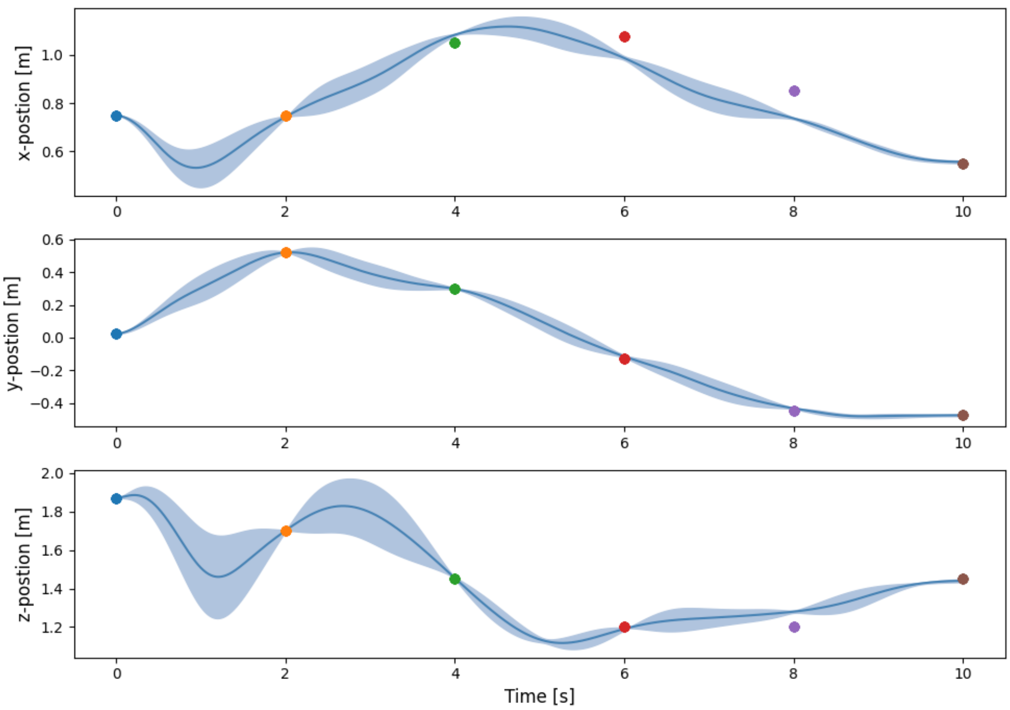


(b) *DMPs 100*

Figure 17: *The images show both the cost functions and the trajectories learned using the adapted method for 100 iterations.*

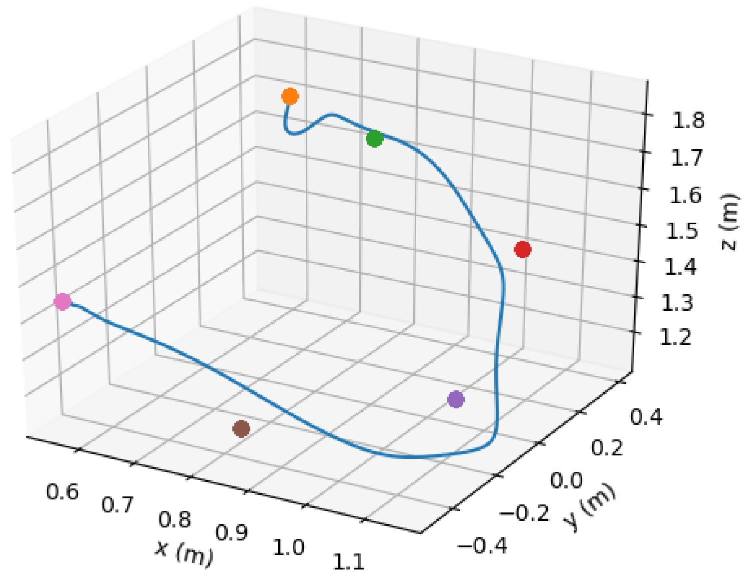


(a) *Cost Function 500*

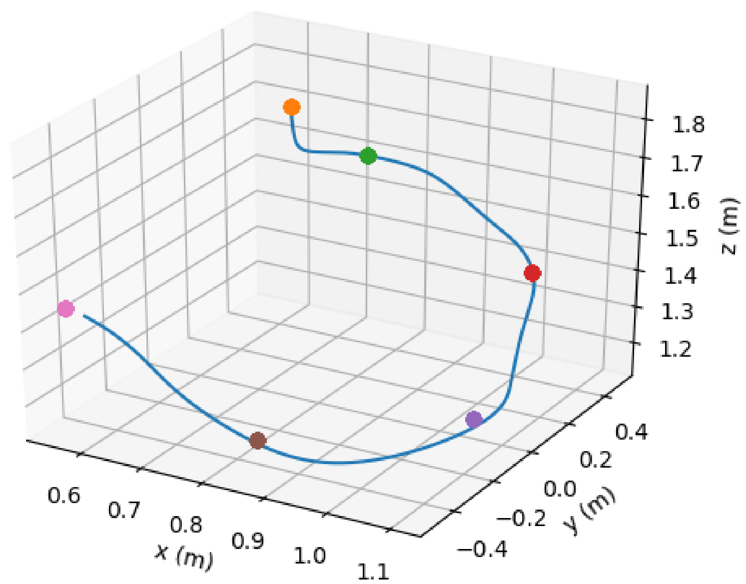


(b) *DMPs 500*

Figure 18: *The images show both the cost functions and the trajectories learned using the adapted method for 500 iterations.*



(a) *Trajectory of an imitated path.*



(b) *Trajectory of the path learned with 100 iterations.*

Figure 19: *This Figure shows two 3D trajectories, **(a)** displays the imitated path and **(b)** a path learned with 100 iterations.*

A.2 Code

In the following section, the most important code of the framework is given. The purpose of this is to provide a deeper understanding of the framework.

A.2.1 Franka Robot

```
import numpy as np
import sympy as sp

# from Franka_ZMQ import FrankaZMQ
from CPS_WS.src.cps_framework.src.Franka_ZMQ import FrankaZMQ


class FrankaRobot:
    def __init__(self, com="ZQM", inverse_controller = "IK"):
        # Communication
        if com == "ZQM":
            self.com = FrankaZMQ()
            self.com_type = com
        elif com == "ROS":
            pass
        else:
            raise ValueError("Communication style not found")
        # Denavit Hartenberg Parameter
        self.a_DH = [0, 0, 0, 0.0825, -0.0825, 0, 0.088, 0]
        self.d_DH = [0.333, 0, 0.316, 0, 0.384, 0, 0, 0.107]
        self.alpha_DH = [0, -sp.pi / 2, sp.pi / 2, sp.pi / 2, -sp.pi / 2,
            sp.pi / 2, sp.pi / 2, 0]

        # Joint constraints
        self.q_max = np.array([2.8973, 1.7628, 2.8973, -0.0698, 2.8973,
           3.7525, 2.8973])
        self.q_min = np.array([-2.8973, -1.7628, -2.8973, -3.0718,
           -2.8973, -0.0175, -2.8973])
        self.dq_max = np.array([2.1750, 2.1750, 2.1750, 2.1750, 2.6100,
           2.6100, 2.6100])
        self.ddq_max = np.array([15, 7.5, 10, 12.5, 15, 20, 20])

        # Configurations Initialization
        self.theta1, self.theta2, self.theta3, self.theta4, self.theta5,
           self.theta6, self.theta7 = \
            sp.symbols('theta_1 theta_2 theta_3 theta_4 theta_5 theta_6
              theta_7', real=True)
        self.theta = sp.Matrix([self.theta1, self.theta2, self.theta3,
                                self.theta4, self.theta5, self.theta6,
                                   self.theta7])

        # region Forward Kinematics Initialization
        self.TE = None  # Transformation matrix end-effector
```

```
        self.pE = None  # Position end-effector
        self.jacobian = None
        # endregion
        self.forward_kinematics_init()

        # Inverse Kinematics
        self._eta_jacobian = 0.1

        # States
        self._theta_state = [0, 0, 0, 0, 0, np.pi/2, np.pi/4]
        self._p_endeffector_state = None
        self.set_initial_pose()

        # Inverse Controller Type
        self._inverse_controller = inverse_controller

    def denavit_hartenberg_matrix(self, dof_nr):
        trans = self.Trans(self.a_DH[dof_nr], 0, self.d_DH[dof_nr])
        rx = self.Rot_x(self.alpha_DH[dof_nr])
        rz = self.Rot_z(self.theta[dof_nr])
        return rx @ trans @ rz

    def forward_kinematics_init(self):
        _p0 = sp.Matrix([0, 0, 0, 1])
        # Transformations
        _T1 = self.denavit_hartenberg_matrix(0)
        _T2 = _T1 * self.denavit_hartenberg_matrix(1)
        _T3 = _T2 * self.denavit_hartenberg_matrix(2)
        _T4 = _T3 * self.denavit_hartenberg_matrix(3)
        _T5 = _T4 * self.denavit_hartenberg_matrix(4)
        _T6 = _T5 * self.denavit_hartenberg_matrix(5)
        _T7 = _T6 * self.denavit_hartenberg_matrix(6)
        self.TE = _T7 * self.Trans(self.a_DH[7], 0, self.d_DH[7])

        self.pE = self.TE * _p0
        self.pE.row_del(3)

        # jacobian
        self.jacobian = self.pE.jacobian(self.theta)

    def forward_kinematics_evaluate(self, theta_eval):
        _pE_eval = self.pE.subs({self.theta1: theta_eval[0],
                                 self.theta2: theta_eval[1],
                                 self.theta3: theta_eval[2],
                                 self.theta4: theta_eval[3],
                                 self.theta5: theta_eval[4],
                                 self.theta6: theta_eval[5],
                                 self.theta7: theta_eval[6]})
        return _pE_eval.evalf()

```

```
    # Jacobian inverse control
    def jacobian_inverse_evaluate(self, theta_eval):
        _jacobian_eval = self.jacobian.subs({self.theta1: theta_eval[0],
                                             self.theta2: theta_eval[1],
                                             self.theta3: theta_eval[2],
                                             self.theta4: theta_eval[3],
                                             self.theta5: theta_eval[4],
                                             self.theta6: theta_eval[5],
                                             self.theta7: theta_eval[6]})

        #print(_jacobian_eval)
        _jacobian_inverse = np.linalg.pinv(_jacobian_eval)
        return _jacobian_inverse.evalf()

    def jacobian_inverse_increment(self, next_goal, recording):

        if self._inverse_controller == "IK":
            _jacobian_inverse = self.jacobian_inverse_evaluate(self.theta_state)
            _delta_pose = self.delta_pose(next_goal)
            _delta_q = np.dot(_jacobian_inverse, _delta_pose)

            _error = 1

            while _error > 0.01:
                _new_q = self.theta_state + self.eta_jacobian * _delta_q

                self._p_endeffector_state = self.forward_kinematics_evaluate(_new_q)
                self._theta_state = _new_q

                _error = np.linalg.norm(self.delta_pose(next_goal))

                self.com.set_joints_values(self._theta_state)

                recording['Time'] = np.append(recording['Time'], self.com.get_simulation_time())

                # add new states and sim time
                if recording['Type'] == "joint":
                    recording['States'] = np.hstack((recording['States'],
                        self.com.get_joint_values()))

                elif recording['Type'] == "task":
                    _new_states = self.com.get_object_position('/Panda_tip')
                    _new_states = _new_states[:, np.newaxis].T
                    recording['States'] = np.vstack((recording['States'],
                        _new_states))

```

```
        elif self._inverse_controller == "CoppeliaSim":

            _error = 1

            while _error > 0.05:
                delta_x = self.compute_delta_x(next_goal)
                _error = np.linalg.norm(delta_x)

                self.move_IK_target(delta_x)

                recording['Time'] = np.append(recording['Time'], self.com.get_simulation_time())

                # add new states and sim time
                if recording['Type'] == "joint":
                    recording['States'] = np.hstack((recording['States'], self.com.get_joint_values()))

                elif recording['Type'] == "task":
                    _new_states = self.com.get_object_position('/Panda_tip')
                    _new_states = _new_states[:, np.newaxis].T
                    recording['States'] = np.vstack((recording['States'], _new_states))

        else:
            raise ValueError('This inverse controller is not implemented!')

        return recording

    def compute_delta_x(self, next_position):
        _current_position_ee = self.com.get_object_position('/Panda_tip')
        return next_position - _current_position_ee

    def move_IK_target(self, delta_x):
        _current_position = self.com.get_object_position('/Panda_target')
        _new_position = _current_position + self.eta_jacobian * delta_x
        self.com.set_object_position('/Panda_target', _new_position)

    def jacobian_inverse_control(self, trajectory, recording_type="task"):
        _time = np.array([0])
        _state_array = None
        if recording_type == "joint":
            _state_array = self.com.get_joint_values()

        elif recording_type == "task":
            _state_array = self.com.get_object_position('/Panda_tip')
            _state_array = _state_array[:, np.newaxis].T
```

```
        else:
            raise ValueError("chosse between 'joint' or 'task' as
               recording type")

        recording = {'Type': recording_type, 'Time': _time, 'States':
           _state_array}

        for t in range(trajectory.shape[1]):
            recording = self.jacobian_inverse_increment(trajectory[:, t],
                recording)

        print('Trajectory finished!')
        return recording

    # Positions
    def reset_target_pose(self):
        # reset Panda Target Position
        if self.com == "ZMQ":
            _current_ee_pose = self.com.get_object_position('/Panda_tip')
            self.com.set_object_position('Panda_target', _current_ee_pose
               )

    def set_initial_pose(self):
        _theta = sp.Matrix(self.theta_state)
        _p_E_state = self.forward_kinematics_evaluate(_theta)

        # Element conversion to Float
        _theta_float = []
        for i in _theta:
            _theta_float.append(float(i))

        self._theta_state = _theta_float
        self._p_endeffector_state = _p_E_state

        self.com.set_joints_values(self._theta_state)

    def delta_pose(self, next_goal):
        _delta_x = next_goal - self.p_endeffector_state
        return _delta_x

    def generate_random_point_in_workspace(self):
        _angles = []
        for i in range(self.q_min.size):
            _rand = np.random.uniform(low=self.q_min[i], high=self.q_max[
               i], size=1)
            _angles.append(_rand[0])

        _pose = self.forward_kinematics_evaluate(_angles)

```

```
            _pose_float = list()
            for i in _pose:
                _pose_float.append(float(i))

            return _pose

        # generate point object at current position in ZMQ
        def gen_current_position_point_object(self, name="/p0"):
            if self.com_type == "ZMQ":
                # transform the endeffector pose to the coordinate system of
                    the baseframe
                _pose = self.transform_to_base_frame(self.
                    _p_endeffector_state)
                _pose_float = self.sympy_to_float(_pose)

                if self.com.sim.getObject(name, {'noError': True}) == -1:
                    self.com.sim.createDummy(0.04)

                    _handle = self.com.sim.getObject('/Dummy')
                    _panda_handle = self.com.sim.getObject('/Panda')

                    self.com.sim.setObjectAlias(_handle, name)
                    self.com.sim.setObjectPosition(_handle, _panda_handle,
                        _pose_float)
                    self.com.sim.setObjectColor(_handle, 0, self.com.sim.
                        colorcomponent_ambient_diffuse, [0., 1., 0.])
                else:
                    self.com.sim.removeObject(self.com.sim.getObjectHandle(
                        name))
                    self.gen_current_position_point_object()

                return _pose_float

            else:
                raise NotImplemented

        # region Spatial Transformations
        @staticmethod
        def Rot_x(phi):
            _R = sp.Matrix([[1, 0, 0, 0],
                            [0, sp.cos(phi), -sp.sin(phi), 0],
                            [0, sp.sin(phi), sp.cos(phi), 0],
                            [0, 0, 0, 1]])
            return _R

        @staticmethod
        def Rot_y(phi):
            _R = sp.Matrix([[sp.cos(phi), 0, -sp.sin(phi), 0],
                            [0, 1, 0, 0],
                            [sp.sin(phi), 0, sp.cos(phi), 0],
```

```
                    [0, 0, 0, 1]])
        return _R

    @staticmethod
    def Rot_z(phi):
        _R = sp.Matrix([[sp.cos(phi), -sp.sin(phi), 0, 0],
                    [sp.sin(phi), sp.cos(phi), 0, 0],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]])
        return _R

    @staticmethod
    def Trans(a, b, c):
        _T = sp.Matrix([[1, 0, 0, a],
                    [0, 1, 0, b],
                    [0, 0, 1, c],
                    [0, 0, 0, 1]])
        return _T

    def transform_to_base_frame(self, vector):
        if isinstance(vector, list):
            vector = np.array(vector)

        vector = np.append(vector, 0)
        _transformed_vector = np.dot(self.Rot_y(np.pi/2)@self.Rot_x(np.pi
            ), vector)

        return _transformed_vector[:3]

    # endregion

    # region Setter Getter
    @property
    def eta_jacobian(self):
        return self._eta_jacobian

    @eta_jacobian.setter
    def eta_jacobian(self, value):
        self._eta_jacobian = value

    @property
    def theta_state(self):
        return self._theta_state

    @property
    def p_endeffector_state(self):
        return self._p_endeffector_state
    # endregion

    # region utility
```

```
    @staticmethod
    def sympy_to_float(vector):
        _vector_float = []
        for i in vector:
            _vector_float.append(float(i))

        return _vector_float

    # endregion


if __name__ == "__main__":
    robo = FrankaRobot()
    print(robo.com.get_joint_values)
```

A.2.2 Franka ZMQ

```
# from zmqRemoteApi import RemoteAPIClient
from CPS_WS.src.cps_framework.src.zmqRemoteApi import RemoteAPIClient
import numpy as np



class FrankaZMQ:
    def __init__(self):
        # client setup
        self.client = RemoteAPIClient('localhost', 23000)
        self.sim = self.client.getObject('sim')

        #  Object handles
        self._panda_base = self.sim.getObjectHandle('/Panda')
        self._panda_tip = self.sim.getObjectHandle('/Panda_tip')

        self._panda_joints = []
        self._nr_joints = 7
        for i in range(1, self._nr_joints+1):
            _joint_name = '/Panda_joint' + str(i)
            self._panda_joints.append(self.sim.getObjectHandle(
               _joint_name))

    # region Setter and Getter
    @property
    def panda_joints(self):
        return self._panda_joints
    # endregion

    def get_joint_values(self):
        _angles = np.zeros((7, 1))
        for idx, i in enumerate(self.panda_joints):
            _angles[idx] = self.sim.getJointPosition(i)
        return _angles
```

```python

    def get_simulation_time(self):
        return self.sim.getSimulationTime()

    def set_joints_values(self, new_angles):
        for i in range(self._nr_joints):
            _joint_handle = self.panda_joints[i]
            _angles_float = float(new_angles[i])
            self.sim.setJointPosition(_joint_handle, _angles_float)

    def set_object_position(self, object_name, new_position,
        relative_frame=-1):
        _object_handle = self.sim.getObjectHandle(object_name)
        self.sim.setObjectPosition(_object_handle, relative_frame,
            new_position.tolist())

    def get_object_matrix(self, object_name, relative_frame=-1):
        _object_handle = self.sim.getObjectHandle(object_name)
        return np.array(self.sim.getObjectMatrix(_object_handle,
            relative_frame))

    def get_object_position(self, object_name, relative_frame=-1):
        _object_handle = self.sim.getObjectHandle(object_name)
        return np.array(self.sim.getObjectPosition(_object_handle,
            relative_frame))

    # generate point object at current position
```

A.2.3 DMP

```python
import numpy as np
import matplotlib.pyplot as plt
from abc import ABC, abstractmethod
import scipy.interpolate as sciip
from sympy import Symbol, diff, exp


class CanonicalSystem:
    def __init__(self, dt, dmp_type='discrete', **kwargs):

        self._dt = dt
        self._x = 1.0

        # get kwargs
        self._a_x = kwargs.get('a_x', 4.0)
        self._tau = kwargs.get('tau', 1.0)

        self._type = dmp_type

        if self._type == 'discrete':
            self._step = self.discrete_time_step
```

```
            self._run_time = 1.0 * self._tau

        elif self._type == 'rhythmic':
            self._step = self.rhythmic_time_step
            self._run_time = 2 * np.pi * self._tau

        else:
            raise ValueError('Pattern has to be either discrete or rhythmic')

        self._time_steps = int(self._run_time / self._dt)
        self._time = np.zeros(self.time_steps)

        self._x_trajectory = np.empty(self.time_steps)
        self._error_coupling = kwargs.get('error_coupling', np.ones(self.time_steps))

        self.roll_out()

    # region CanonicalSystem methods
    def reset_state(self):
        self._x = 1.0
        self._time = np.zeros(self.time_steps)

    def discrete_time_step(self, error_coupling=1.0, time_index=0):
        self._x *= np.exp((-self._a_x * error_coupling / self._tau) * self._dt)
        # self._x += (-self._a_x * self._x * error_coupling) / self._tau * self._dt

        if time_index == 0:
            self._time[time_index] = 0
        elif time_index != 0:
            self._time[time_index] = self.time[time_index - 1] + self._dt
        return self.x

    def rhythmic_time_step(self, error_coupling=1.0, time_index=0):
        self._x += self._dt * error_coupling / self._tau

        if time_index == 0:
            self._time[time_index] = 0
        elif time_index != 0:
            self._time[time_index] = self.time[time_index - 1] + self._dt
        return self.x

    def roll_out(self):
        self.reset_state()

        for i in range(self.time_steps):
            self._x_trajectory[i] = self.x
```

```
                self._step(self._error_coupling[i], i)

        return self._x_trajectory

    # endregion

    # region Getter Setter
    @property
    def x(self):
        return self._x

    @property
    def x_trajectory(self):
        return self._x_trajectory

    @property
    def time(self):
        return self._time

    @property
    def step(self):
        return self._step

    @property
    def time_steps(self):
        return self._time_steps

    @property
    def run_time(self):
        return self._run_time

    @property
    def a_x(self):
        return self._a_x
    # endregion


class DMP(ABC):
    def __init__(self, nr_dmps, nr_bfs, dt=0.01, **kwargs):
        self._nr_dmps = nr_dmps
        self._nr_bfs = nr_bfs
        self._dt = dt

        self._vector_size = (1, self.nr_dmps)

        # get start and goal points
        self._y0 = kwargs.get('y0', np.zeros(self._vector_size))
        self._g = kwargs.get('goal', np.ones(self._vector_size))
        self._gdy = kwargs.get('goal_dy', np.zeros(self._vector_size))

```

```
        # weights generation
        self._w_gen = kwargs.get('w_gen', 'zeros')
        self._w = kwargs.get('w', self.reset_weigth())

        # get important params
        self._a_z = kwargs.get('a_z', 25 * np.ones(self._vector_size))
        self._b_z = kwargs.get('b_z', self._a_z / 4)
        self._tau = kwargs.get('tau', 1.0)

        # initialize canonical system
        _a_x = float(self._a_z[:, 0] / 3)
        self._cs = CanonicalSystem(dt=self._dt, a_x=_a_x, **kwargs)
        self._time_steps = self.cs.time_steps

        # initialize state vectors of
        self._y = self._y0.copy()
        self._dy = np.zeros(self._vector_size)
        self._ddy = np.zeros(self._vector_size)

        # check dimensions and offset
        self._dimension_checker()
        self._offset_checker()

        # imitation learning
        self.y_des = None

    # region DMP methods
    def _generate_start(self, y_des):
        _start = np.zeros(self._vector_size)
        for dim in range(self.nr_dmps):
            _start[:, dim] = y_des[0, dim]

        return _start

    def reset_weigth(self):
        self.random_gen = np.random.default_rng()
        if self._w_gen == 'zeros':
            _w = np.zeros((self.nr_dmps, self.nr_bfs))

        elif self._w_gen == 'random':
            _w = 200 * self.random_gen.random((self.nr_dmps, self.nr_bfs)
                ) - 100
        else:
            raise ValueError('weight generations can be zero or random')

        self._w = _w

        return self._w

    def reset_states(self):
```

```
        self.cs.reset_state()

        self._y = self._y0.copy()
        self._dy = np.zeros(self._vector_size)
        self._ddy = np.zeros(self._vector_size)

    def _dimension_checker(self):
        if self.y0.shape[1] != self._nr_dmps or self.y0.shape[0] != 1:
            raise ValueError('y0 needs the shape [nr_dmps, 1]')

        if self.g.shape[1] != self._nr_dmps or self.y0.shape[0] != 1:
            raise ValueError('g needs the shape [nr_dmps, 1]')

    def _offset_checker(self):
        for i in range(self._nr_dmps):
            if abs(self.y0[:, i] - self.g[:, i]) < 1e-4:
                self.g[i] += 1e-4

    def step(self, error=0.0, spatial_coupling=None, time_index=0):
        # step in canonical system
        _error_coupling = 1.0 / (1.0 + error)
        _x = self.cs.step(error_coupling=_error_coupling, time_index=time_index)

        # initialise basis function
        _psi, _sum_psi = self._generate_psi(_x)
        for dim in range(self.nr_dmps):
            f = self._generate_front_term(_x, dim) * (np.dot(_psi, self._w[dim, :]))
            f /= _sum_psi

            # self._ddy[:, dim] = self._a_z[:, dim] * \
            #                     ((self._b_z[:, dim] * (self.g[:, dim] - self._y[:, dim])) -
            #                      (-self.gdy[:, dim] + self._dy[:, dim])) + f

            self._ddy[:, dim] = self._a_z[:, dim] * \
                                ((self._b_z[:, dim] * (self.g[:, dim] - self._y[:, dim])) - self._dy[:, dim]) + f

            if spatial_coupling is not None:
                self._ddy[:, dim] += spatial_coupling[dim]

            self._dy[:, dim] += self._ddy[:, dim] / self._tau * self._dt * _error_coupling
            self._y[:, dim] += self._dy[:, dim] * self._dt / self._tau * _error_coupling

```

```
        return self._y, self._dy, self._ddy

    def roll_out(self, **kwargs):
        self.reset_states()

        _y_trajectory = np.zeros((self._time_steps, self.nr_dmps))
        _dy_trajectory = np.zeros((self._time_steps, self.nr_dmps))
        _ddy_trajectory = np.zeros((self._time_steps, self.nr_dmps))

        for t in range(self.cs.time_steps):
            _y_trajectory[t, :], _dy_trajectory[t, :], _ddy_trajectory[t,
                :] = self.step(time_index=t, **kwargs)

        return _y_trajectory, _dy_trajectory, _ddy_trajectory

    def _interpolate_path(self, y_des):
        _path = np.zeros((self._time_steps, self.nr_dmps))
        _x = np.linspace(0, self.cs.run_time, y_des.shape[0])

        for dim in range(self.nr_dmps):
            _path_generation = sciip.interp1d(_x, y_des[:, dim])
            for t in range(self._time_steps):
                _path[t, dim] = _path_generation(t * self._dt)

        return _path

    def imitation_learning(self, y_des):
        if y_des.shape[1] != self.nr_dmps:
            raise ValueError('y_des needs the shape [nr_dmps, selectable
                ]!')

        if y_des.ndim == 1:
            y_des = y_des.reshape(self._vector_size)

        self._y0 = self._generate_start(y_des)
        self._g = self._generate_goal(y_des)

        _y_des = self._interpolate_path(y_des)
        _dy_des = np.gradient(_y_des, axis=0) / self._dt
        _ddy_des = np.gradient(_dy_des, axis=0) / self._dt

        self.y_des = _y_des.copy()

        _f_target = np.zeros((self._time_steps, self.nr_dmps))
        for dim in range(self.nr_dmps):
            _f_target[:, dim] = self._tau ** 2 * _ddy_des[:, dim] - \
                                self._a_z[:, dim] * (self._b_z[:, dim] *
                                                     (self.g[:, dim] -
                                                        _y_des[:, dim]) -
                                                         self._tau *
```

```
                                                    _dy_des[:, dim])

        self._generate_weights(_f_target)

    # endregion

    # region abstract methods
    @abstractmethod
    def _generate_front_term(self, x, dmp_index):
        pass

    @abstractmethod
    def _generate_goal(self, y_des):
        pass

    @abstractmethod
    def _generate_psi(self, x):
        pass

    @abstractmethod
    def _generate_weights(self, f_target):
        pass

    # endregion

    # region Getter and Setter
    @property
    def y0(self):
        return self._y0

    @property
    def g(self):
        return self._g

    @property
    def nr_bfs(self):
        return self._nr_bfs

    @property
    def nr_dmps(self):
        return self._nr_dmps

    @property
    def cs(self):
        return self._cs

    @property
    def w(self):
        return self._w

```

```
    @property
    def gdy(self):
        return self._gdy

    @w.setter
    def w(self, value):
        if value.shape == (self.nr_dmps, self.nr_bfs):
            self._w = value

        elif value.shape == (self.nr_dmps * self.nr_bfs, None) or (self.nr_dmps * self.nr_bfs, 1):
            value = np.reshape(value, (self.nr_dmps, self.nr_bfs))
            self._w = value
        else:
            raise ValueError('w needs shape [self.nr_dmps, self.nr_bfs] or [self.nr_dmps * self.nr_bfs, None]')

    @y0.setter
    def y0(self, value):
        if value.shape == (self.nr_dmps,) or (self.nr_dmps, 1):
            self._y0 = value
        else:
            raise ValueError('y0 needs shape (nr_dmps,) or (nr_dmps, 1)')

    @g.setter
    def g(self, value):
        if value.shape == (self.nr_dmps,) or (self.nr_dmps, 1):
            self._g = value
        else:
            raise ValueError('g needs shape (nr_dmps,) or (nr_dmps, 1)')
    # endregion


class DmpDiscrete(DMP):
    def __init__(self, **kwargs):
        super(DmpDiscrete, self).__init__(pattern='discrete', **kwargs)

        # discrete dmp initialization
        self._c = np.zeros((self.nr_bfs, 1))
        self._h = np.zeros((self.nr_bfs, 1))
        self._generate_basis_function_parameters()

        # imitation learning
        self._regression_type = kwargs.get('regression_type', 'Schaal')
        self._imitation_type = kwargs.get('imitation_type', 'eye')
        self._reg_lambda = kwargs.get('reg_lambda', 1e-12)

        # psi
        self._psi, _ = self._generate_psi(self.cs.roll_out())

```

```
    def _generate_basis_function_parameters(self):
        for i in range(1, self.nr_bfs + 1):
            _des_c = (i - 1) / (self.nr_bfs - 1)
            self._c[i - 1] = np.exp(-self.cs.a_x * _des_c)

        for i in range(self.nr_bfs):
            if i != self.nr_bfs - 1:
                self._h[i] = (self.c[i + 1] - self.c[i]) ** (-2)
            else:
                self._h[i] = self._h[i - 1]

    def _generate_front_term(self, x, dmp_index=None):
        if dmp_index is not None:
            _s = x * (self.g[:, dmp_index] - self.y0[:, dmp_index])

        else:
            _s = np.zeros((self.cs.time_steps, self.nr_dmps))
            for dim in range(self.nr_dmps):
                _s[:, dim] = x * (self.g[:, dim] - self.y0[:, dim])

        return _s

    def _generate_goal(self, y_des):
        _goal = np.ones(self._vector_size)
        for dim in range(self.nr_dmps):
            _goal[:, dim] = y_des[-1, dim]

        return _goal

    def _generate_psi(self, x):
        _psi = (np.exp(-self.h * (x - self.c) ** 2)).T

        if x.shape == ():
            _sum_psi = np.sum(_psi)
            return _psi, _sum_psi

        elif x.shape[0] == self.cs.time_steps:
            _sum_psi = np.sum(_psi, axis=1)
            return _psi, _sum_psi

    def psi_plot(self, x):
        _s = self._generate_front_term(x)
        _psi = (np.exp(-self.h * (x - self.c) ** 2)).T
        _sum_psi = _sum_psi = np.sum(_psi, axis=1)

        _psi_activations = np.zeros((_psi.shape[0], _psi.shape[1], _s.
            shape[1]))
        print(_s.shape)
        for t in range(x.shape[0]):
            for dim in range(_s.shape[1]):
```

```
                _psi_activations[t, :, dim] = _psi[t, :] / _sum_psi[t]
                _psi_activations[t, :, dim] *= _s[t, dim]

        return _psi_activations

    def generate_psi_3rd_derivative(self, x):
        _, _psi_sum = self._generate_psi(x)

        _x = Symbol('x')
        _h = Symbol('h')
        _c = Symbol('c')
        _g = Symbol('g')
        _y = Symbol('y')

        _psi = exp(-_h * (_x - _c) ** 2)
        _s = _x * (_g - _y)
        func = _psi * _s

        _psi_dev = np.zeros((self.cs.time_steps, self.nr_bfs, self.nr_dmps))

        for dim in range(self.nr_dmps):
            for bf in range(self.nr_bfs):
                for idx, t in enumerate(x):
                    psi_eval = func.evalf(subs={_h: float(self.h[bf, :]),
                                                _c: float(self._c[bf, :]),
                                                _g: float(self.g[:, dim]),
                                                _y: float(self.y0[:, dim])})

                    psi_diff = diff(psi_eval, _x, 3)
                    _psi_dev[idx, bf, dim] = psi_diff.evalf(subs={_x: t}) / _psi_sum[idx]

        return _psi_dev

    def _generate_weights(self, f_target):
        _x_trajectory = self.cs.roll_out()
        _psi, _sum_psi = self._generate_psi(_x_trajectory)

        _s = self._generate_front_term(_x_trajectory)
        _sT = _s.T

        if self._regression_type == 'Schaal':

            for dim in range(self.nr_dmps):
                _k = self.g[:, dim] - self.y0[:, dim]
                for bf in range(self.nr_bfs):
```

```
                    self._w[dim, bf] = np.dot(np.dot(_sT[dim, :], np.diag
                        (_psi[:, bf])), f_target[:, dim]) / \
                                        (np.dot(np.dot(_sT[dim, :], np.
                                            diag(_psi[:, bf])), _s[:, dim])
                                            )

            self._w = np.nan_to_num(self._w)

        elif self._regression_type == 'RidgeRegression':
            _regression_matrix = np.zeros((self.nr_bfs, self.nr_bfs, self
                .nr_dmps))

            if self._imitation_type == 'eye':
                for dim in range(self.nr_dmps):
                    _regression_matrix[:, :, dim] = np.eye(self.nr_bfs)

            elif self._imitation_type == 'jerk':
                _Gamma = self.generate_psi_3rd_derivative(_x_trajectory)
                for dim in range(self.nr_dmps):
                    _regression_matrix[:, :, dim] = _Gamma[:, :, dim].T @
                        _Gamma[:, :, dim]

            else:
                raise ValueError('Imitation_type can either be eye or
                    jerk')

            _psi_new = np.zeros((_psi.shape[0], _psi.shape[1], self.
                nr_dmps))
            for dim in range(self.nr_dmps):
                for bf in range(self.nr_bfs):
                    _psi_new[:, bf, dim] = _psi[:, bf] / _sum_psi * _s[:,
                        dim]

            for dim in range(self.nr_dmps):
                _matrix = np.linalg.inv(_psi_new[:, :, dim].T @ _psi_new
                    [:, :, dim] + \
                                        self._reg_lambda *
                                            _regression_matrix[:, :, dim])
                                            @ _psi_new[:, :, dim].T
                self._w[dim, :] = np.dot(_matrix, f_target[:, dim])

            self._w = np.nan_to_num(self._w)

    # region Getter and Setter
    @property
    def c(self):
        return self._c

    @property
    def h(self):
```

```
        return self._h

    @property
    def psi(self):
        return self._psi

    # endregion


if __name__ == '__main__':

    bfs = 20
    dmp = DmpDiscrete(nr_dmps=2, nr_bfs=bfs, dt=0.001, regression_type='
        RidgeRegression', reg_lambda=0.5 * 1e-5)

    dmp.cs.roll_out()
    psi_activations = dmp.psi_plot(dmp.cs.x_trajectory)

    plt.figure(1, figsize=(10, 3))
    plt.plot(dmp.cs.time, dmp.cs.x_trajectory)
    plt.xlabel("time (s)")
    plt.ylabel("x value")
    plt.tight_layout()
    plt.savefig('CanonicalSystem.png')

    plt.figure(2, figsize=(10, 3))

    # plt.subplot(211)
    for i in range(dmp.nr_bfs):
        plt.plot(dmp.cs.time, dmp.psi[:, i])
    plt.xlabel("time (s)")
    plt.ylabel("activation")
    plt.tight_layout()
    plt.savefig('Psi.png')

    plt.figure(3, figsize=(10, 3))
    #plt.subplot(212)
    for i in range(dmp.nr_bfs):
        plt.plot(dmp.cs.time, psi_activations[:, i, 0])
    plt.xlabel("time (s)")
    plt.ylabel("activation")
    plt.tight_layout()
    plt.savefig('PsiScaled.png')

    # a straight line to target
    path1 = np.sin(np.arange(0, 1, 0.01) * 5)
    # a strange path to target
    path2 = np.zeros(path1.shape)
    path2[int(len(path2) / 2.0):] = 0.5

```

```python
    dmp.imitation_learning(y_des=np.array([path1, path2]).T)
    # change the scale of the movement
    dmp.g[0, 0] = 3
    dmp.g[0, 1] = 2

    y_track, dy_track, ddy_track = dmp.roll_out()

    plt.figure(4, figsize=(10, 6))
    plt.subplot(211)
    plt.plot(y_track[:, 0], lw=2)
    plt.subplot(212)
    plt.plot(y_track[:, 1], lw=2)

    plt.subplot(211)
    a = plt.plot(dmp.y_des[:, 0] / path1[-1] * dmp.g[:, 0], "r--", lw=2)
    plt.title("x-coordinate")
    plt.xlabel("time (ms)")
    plt.ylabel("system trajectory")
    plt.legend(['generated path', 'desired path'], loc="lower right")
    plt.subplot(212)
    b = plt.plot(dmp.y_des[:, 1] / path2[-1] * dmp.g[:, 1], "r--", lw=2)
    plt.title("y-coordinate")
    plt.xlabel("time (ms)")
    plt.ylabel("system trajectory")
    plt.legend(['generated path', 'desired path'], loc="lower right")
    plt.tight_layout()
    plt.savefig('ImitationLearning.png')

    plt.show()
```

A.2.4 Experiment Reinforcement Learnig

```python
from CPS_WS.src.cps_framework.src.RL_standard import CpsRl
import numpy as np
import time
import matplotlib.pyplot as plt
from CPS_WS.src.cps_framework.src.Franka_Robot import FrankaRobot
from csv import DictWriter
from datetime import datetime


def write_weights(nr_dmps, nr_bfs, scaling, nr_iteration, tau, weights, rewards, compute_time, way_points, filename='scaling_weights.csv'):
    with open(filename, 'a+', newline='') as write_obj:
        now = datetime.now()
        field_names = ['Time', 'NrDmps', 'NrBfs', 'Scaling', 'Tau', 'NrIterations',
                       'Weights', 'Rewards', 'ComputeTime', 'NrWayPoints', 'WayPoints']
        new_data = {'Time': now, 'NrDmps': nr_dmps, 'NrBfs': nr_bfs, 'Scaling': scaling, 'Tau': tau,
```

```
                        'NrIterations': nr_iteration, 'Weights': weights.
                            tolist(), 'Rewards': rewards.tolist(), '
                            ComputeTime': compute_time,
                        'NrWayPoints': way_points.shape[1], 'WayPoints':
                            way_points.tolist()}

        dict_writer = DictWriter(write_obj, fieldnames=field_names)
        #dict_writer.writeheader()
        dict_writer.writerow(new_data)


def main():
    # Franka initialization
    print('Initialize Franka')
    robot = FrankaRobot(inverse_controller="CoppeliaSim")

    # Task Space Learning

    # DMP Parameters
    nr_dmps = 3  # x,y,z
    nr_bfs = 25 # number of basis functions
    tau = 10.0  # time scaling variable
    goal = robot.com.get_object_position('/pT')
    goal = goal[:, np.newaxis].T
    y0 = robot.com.get_object_position('/p0')
    y0 = y0[:, np.newaxis].T

    # RL Parameters and cost functional parameters
    max_itr = [200, 200, 200, 200, 200, 200, 200, 200, 200, 200]
    scaling = [True, True, True, True, True, False, False, False, False,
        False]
    goal_penalty = 1e5
    via_point_penalty = 1e5
    velocity_penalty = 1e3
    acceleration_penalty = 1e-2
    print('Ready')

    # Via Points
    via_point_names = ['/p1', '/p2', '/p3', '/p4']
    via_point_timing = [0.2 * tau, 0.4 * tau, 0.6 * tau, 0.8 * tau]
    via_points = np.zeros((nr_dmps + 1, len(via_point_names)))

    # Generate Point array
    way_points = np.zeros((nr_dmps + 1, len(via_point_names) + 2))
    way_points[:3, 0] = y0
    way_points[:3, -1] = goal
    way_points[3, -1] = 1.0 * tau

    for i, via in enumerate(via_point_names):
        _temp = robot.com.get_object_position(via)
```

```
        _temp = np.append(_temp, via_point_timing[i])
        via_points[:, i] = _temp
        way_points[:, i+1] = _temp

    for iteration in range(len(max_itr)):
        print('Start Iteration Number : {}'.format(iteration))
        # Initialize RL
        rl = CpsRl(nr_dmps=nr_dmps,
                   nr_bfs=nr_bfs,
                   tau=tau,
                   y0=y0,
                   goal=goal,
                   canonical_time=True,
                   VarMin=-1e2,
                   VarMax=1e2,
                   MaxIt=max_itr[iteration],
                   scaling=scaling[iteration],
                   via_points=via_points,
                   via_penalty=via_point_penalty,
                   goal_penalty=goal_penalty,
                   velo_penalty=velocity_penalty,
                   accelerate_penalty=acceleration_penalty)

        # Start timer
        tic = time.perf_counter()

        # RL runner
        while not rl.cma.stop():
            rl.runner()

        # Stop timer
        toc = time.perf_counter()

        # evaluate best solution
        final_weights = rl.cma.BestSol["Position"].reshape(rl.dmp.w.shape
            )
        rl.dmp.w = (final_weights * rl.weight_scale_array)

        y_track, dy_track, ddy_track = rl.dmp.roll_out()

        # plot the results
        time_scale_plotting = rl.dmp.cs.time_steps / tau

        plt.figure(iteration)

        # plot of start, goal and via-points in each dimension
        # x - coordinate
        plt.subplot(311)

        plt.plot(y_track[:, 0], lw=2)
```

```
        plt.plot(rl.dmp.cs.time[0] * time_scale_plotting, y0[:, 0], 'o')
        plt.plot(rl.dmp.cs.time[-1] * time_scale_plotting, goal[:, 0], 'o
            ')
        for i in range(via_points.shape[1]):
            plt.plot(via_points[-1, i] * time_scale_plotting, via_points
                [0, i], 'o')

        plt.title("Number of Basis functions = {}, scaling ={}".format(
            nr_bfs, scaling[iteration]))

        # y - coordinate
        plt.subplot(312)

        plt.plot(y_track[:, 1], lw=2)

        plt.plot(rl.dmp.cs.time[0] * time_scale_plotting, y0[:, 1], 'o')
        plt.plot(rl.dmp.cs.time[-1] * time_scale_plotting, goal[:, 1], 'o
            ')
        for i in range(via_points.shape[1]):
            plt.plot(via_points[-1, i] * time_scale_plotting, via_points
                [1, i], 'o')

        # z - coordinate
        plt.subplot(313)

        plt.plot(y_track[:, 2], lw=2)

        plt.plot(rl.dmp.cs.time[0] * time_scale_plotting, y0[:, 2], 'o')
        plt.plot(rl.dmp.cs.time[-1] * time_scale_plotting, goal[:, 2], 'o
            ')
        for i in range(via_points.shape[1]):
            plt.plot(via_points[-1, i] * time_scale_plotting, via_points
                [2, i], 'o')
        plt.tight_layout()
        plt.savefig('Scaling/DMP' + str(iteration) + '.png')

        # Reward Plot

        fig2 = plt.figure(2*iteration+1)
        ax2 = fig2.add_subplot()
        iterations = np.linspace(0, rl.cma.itr, rl.cma.itr)
        ax2.plot(iterations, rl.cma.BestCost)

        ax2.set_yscale('log')
        plt.title("Max Iterations = {}, scaling ={}".format(max_itr[
            iteration], scaling[iteration]))
        plt.savefig('Scaling/Reward' + str(iteration) + '.png')

        plt.show()
```

```python

        write_weights(nr_dmps, nr_bfs, scaling[iteration], max_itr[iteration], tau, (final_weights * rl.weight_scale_array),
                      rl.cma.BestCost.T, toc - tic, way_points, 'Scaling/scaling.csv')

        print(f"Computing time of the reinforcement algorithm: {(toc - tic) / 60} minutes!")


if __name__ == "__main__":
    main()
```

A.2.5 Experiment Imitation Learning

```python
from CPS_WS.src.cps_framework.src.RL_standard import CpsRl
import numpy as np
import time
import matplotlib.pyplot as plt
from CPS_WS.src.cps_framework.src.Franka_Robot import FrankaRobot
from csv import DictWriter
from datetime import datetime


rnd_gen = np.random.default_rng()


def write_weights(nr_dmps, nr_bfs, scaling, nr_iteration, tau, weights, rewards, compute_time, way_points, filename='imitation_weights.csv'):
    with open(filename, 'a+', newline='') as write_obj:
        now = datetime.now()
        field_names = ['Time', 'NrDmps', 'NrBfs', 'Scaling', 'Tau', 'NrIterations',
                       'Weights', 'Rewards', 'ComputeTime', 'NrWayPoints', 'WayPoints']
        new_data = {'Time': now, 'NrDmps': nr_dmps, 'NrBfs': nr_bfs, 'Scaling': scaling, 'Tau': tau,
                    'NrIterations': nr_iteration, 'Weights': weights.tolist(), 'Rewards': rewards.tolist(), 'ComputeTime': compute_time,
                    'NrWayPoints': way_points.shape[1], 'WayPoints': way_points.tolist()}

        dict_writer = DictWriter(write_obj, fieldnames=field_names)
        #dict_writer.writeheader()
        dict_writer.writerow(new_data)


def vary_via_points(via_points_array):
    _mean = 0
    _sigma = 0.05
```

```
    for i in range(via_points_array.shape[1]):
        _x, _y, _z, _t = via_points_array[:, i]
        _x += rnd_gen.normal(_mean, _sigma)
        _y += rnd_gen.normal(_mean, _sigma)
        _z += rnd_gen.normal(_mean, _sigma)
        via_points_array[:, i] = np.array([_x, _y, _z, _t])

    return via_points_array


def main():
    # Franka initialization
    print('init')
    robot = FrankaRobot(inverse_controller="CoppeliaSim")

    # Task Space Learning

    # DMP Parameters
    nr_dmps = 3  # x,y,z
    nr_bfs = 25  # number of basis functions
    tau = 10.0   # time scaling variable
    goal = robot.com.get_object_position('/pT')
    goal = goal[:, np.newaxis].T
    y0 = robot.com.get_object_position('/p0')
    y0 = y0[:, np.newaxis].T

    # RL Parameters and cost functional parameters
    max_itr = [200, 200, 200, 200, 200, 200, 200, 200, 200, 200]
    goal_penalty = 1e5
    via_point_penalty = 1e5
    velocity_penalty = 1e3
    acceleration_penalty = 1e-2
    print('Ready')

    # Via Points
    via_point_names = ['/p1', '/p2', '/p3', '/p4']
    via_point_timing = [0.2 * tau, 0.4 * tau, 0.6 * tau, 0.8 * tau]
    via_points_fix = np.zeros((nr_dmps + 1, len(via_point_names)))

    # Generate Point array
    way_points = np.zeros((nr_dmps + 1, len(via_point_names) + 2))
    way_points[:3, 0] = y0
    way_points[:3, -1] = goal
    way_points[3, -1] = 1.0 * tau

    for i, via in enumerate(via_point_names):
        _temp = robot.com.get_object_position(via)
        _temp = np.append(_temp, via_point_timing[i])
        via_points_fix[:, i] = _temp
        way_points[:, i + 1] = _temp
```

```python

    for iteration in range(len(max_itr)):
        print('Start Iteration Number : {}'.format(iteration))
        via_points = vary_via_points(via_points_fix)
        way_points[:, 1:5] = via_points
        # Initialize RL
        rl = CpsRl(nr_dmps=nr_dmps,
                   nr_bfs=nr_bfs,
                   tau=tau,
                   y0=y0,
                   goal=goal,
                   canonical_time=True,
                   VarMin=-1e2,
                   VarMax=1e2,
                   MaxIt=max_itr[iteration],
                   scaling=False,
                   via_points=via_points,
                   via_penalty=via_point_penalty,
                   goal_penalty=goal_penalty,
                   velo_penalty=velocity_penalty,
                   accelerate_penalty=acceleration_penalty)

        # Start timer
        tic = time.perf_counter()

        # RL runner
        while not rl.cma.stop():
            rl.runner()

        # Stop timer
        toc = time.perf_counter()

        # evaluate best solution
        final_weights = rl.cma.BestSol["Position"].reshape(rl.dmp.w.shape)
        rl.dmp.w = (final_weights * rl.weight_scale_array)

        y_track, dy_track, ddy_track = rl.dmp.roll_out()

        # plot the results
        time_scale_plotting = rl.dmp.cs.time_steps / tau

        plt.figure(iteration)

        # plot of start, goal and via-points in each dimension
        # x - coordinate
        plt.subplot(311)

        plt.plot(y_track[:, 0], lw=2)

```

```
        plt.plot(rl.dmp.cs.time[0] * time_scale_plotting, y0[:, 0], 'o')
        plt.plot(rl.dmp.cs.time[-1] * time_scale_plotting, goal[:, 0], 'o')
        for i in range(via_points.shape[1]):
            plt.plot(via_points[-1, i] * time_scale_plotting, via_points[0, i], 'o')

        plt.title("Number of Basis functions = {}".format(nr_bfs))

        # y - coordinate
        plt.subplot(312)

        plt.plot(y_track[:, 1], lw=2)

        plt.plot(rl.dmp.cs.time[0] * time_scale_plotting, y0[:, 1], 'o')
        plt.plot(rl.dmp.cs.time[-1] * time_scale_plotting, goal[:, 1], 'o')
        for i in range(via_points.shape[1]):
            plt.plot(via_points[-1, i] * time_scale_plotting, via_points[1, i], 'o')

        # z - coordinate
        plt.subplot(313)

        plt.plot(y_track[:, 2], lw=2)

        plt.plot(rl.dmp.cs.time[0] * time_scale_plotting, y0[:, 2], 'o')
        plt.plot(rl.dmp.cs.time[-1] * time_scale_plotting, goal[:, 2], 'o')
        for i in range(via_points.shape[1]):
            plt.plot(via_points[-1, i] * time_scale_plotting, via_points[2, i], 'o')
        plt.tight_layout()
        plt.savefig('Imitation/DMP' + str(iteration) + '.png')

        # Reward Plot

        fig2 = plt.figure(2 * iteration + 1)
        ax2 = fig2.add_subplot()
        iterations = np.linspace(0, rl.cma.itr, rl.cma.itr)
        ax2.plot(iterations, rl.cma.BestCost)

        ax2.set_yscale('log')
        plt.title("Max Iterations = {}".format(max_itr[iteration]))
        plt.savefig('Imitation/Reward' + str(iteration) + '.png')

        plt.show()

        write_weights(nr_dmps, nr_bfs, False, max_itr[iteration], tau,
                      (final_weights * rl.weight_scale_array),
```

```
                        rl.cma.BestCost.T, toc - tic, way_points, 'Imitation/weights.csv')

        print(f"Computing time of the reinforcement algorithm: {(toc - tic) / 60} minutes!")


if __name__ == "__main__":
    main()
```

A.2.6 Evalutaion

```
import csv
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits import mplot3d

# from DMP import DmpDiscrete
# from Franka_Robot import FrankaRobot
from CPS_WS.src.cps_framework.src.DMP import DmpDiscrete
from CPS_WS.src.cps_framework.src.Franka_Robot import FrankaRobot


def import_weight_data(filename='../src/Scaling/scaling.csv'):
    with open(filename, 'r') as csvfile:
        reader = csv.DictReader(csvfile)
        # headers = reader.fieldnames

        # get array information from the csv
        _nr_dmp = []
        _nr_bfs = []
        _nr_iteration = []
        _tau = []
        _scaled = []
        _rewards_list = []
        _weights_list = []
        _waypoint_list = []
        _nr_waypoints = []
        _nr_lines = 0
        _nr_scaled = 0
        _nr_non_scaled = 0
        for _row in reader:
            _nr_dmp.append(int(_row['NrDmps']))
            _nr_bfs.append(int(_row['NrBfs']))
            _nr_iteration.append(int(_row['NrIterations']))
            _tau.append(float(_row['Tau']))
            if _row['Scaling'] == 'True':
                _scaled.append(True)
                _nr_scaled += 1
            else:
                _scaled.append(False)
```

```
                _nr_non_scaled += 1

            _nr_waypoints.append(int(_row['NrWayPoints']))

            _rewards_list.append(_row['Rewards'])
            _weights_list.append(_row['Weights'])
            _waypoint_list.append(_row['WayPoints'])

            _nr_lines += 1

        # Export Weights, Rewards and Waypoints to Numpy arrays

        # init Numpy arrays
        _rewards = np.zeros([_nr_lines, _nr_iteration[0] - 1])
        _weights = np.zeros([_nr_dmp[0], _nr_bfs[0], _nr_lines])
        _way_points = np.zeros([_nr_dmp[0] + 1, _nr_waypoints[0],
            _nr_lines])

        # Iteration over all lines
        for i in range(_nr_lines):
            _reward_str = _rewards_list[i].strip('[]')
            _rewards[i, :] = np.fromstring(_reward_str, dtype=np.float32,
                sep=',')

            _weight_str = _weights_list[i].strip('[]')
            _weight_str = _weight_str.replace('[', '').replace(']', '')
            _weight = np.fromstring(_weight_str, dtype=np.float32, sep=',
                ')
            _weight = _weight.reshape((_nr_dmp[i], _nr_bfs[i]))
            _weights[:, :, i] = _weight

            _way_point_str = _waypoint_list[i].replace('[', '').replace('
                ]', '')
            _way_point = np.fromstring(_way_point_str, dtype=np.float32,
                sep=',')
            _way_point = _way_point.reshape((_nr_dmp[i] + 1,
                _nr_waypoints[i]))
            _way_points[:, :, i] = _way_point

        temp_dict = {'nr_dmps': _nr_dmp, 'nr_bfs': _nr_bfs, '
            nr_iterations': _nr_iteration, 'scaled': _scaled,
                     'nr_of_scaled': _nr_scaled, 'nr_of_non_scaled':
                         _nr_non_scaled,
                     'tau': _tau, 'weights': _weights, 'rewards':
                         _rewards,
                     'nr_way_points': _nr_waypoints, 'way_points':
                         _way_points}

        return temp_dict

```

```python

def read_recordings(filename='Imitation/recordings.csv'):
    with open(filename, 'r') as read_obj:
        reader = csv.DictReader(read_obj)

        _recording_list = []
        for _row in reader:
            _recording = {}
            _recording['Type'] = _row['Type']

            _time_str = _row['Time']
            _time_str = _time_str.replace('[', '').replace(']', '')
            _recording['Time'] = np.fromstring(_time_str, dtype=np.float32, sep=',')

            _trajectory_str = _row['States']
            _trajectory_str = _trajectory_str.replace('[', '').replace(']', '')
            _trajectory = np.fromstring(_trajectory_str, dtype=np.float32, sep=',')
            _recording['States'] = _trajectory.reshape(_recording['Time'].shape[0], int(_row['NrDmps']))

            _recording_list.append(_recording)

    return _recording_list


def write_recordings(recoring_list, filename='Imitation/recordings.csv'):
    with open(filename, 'a+', newline='') as write_obj:
        field_names = ['Type', 'Time', 'States', 'NrDmps']

        dict_writer = csv.DictWriter(write_obj, fieldnames=field_names)
        dict_writer.writeheader()

        for i in range(len(recoring_list)):
            new_data = {'Type': recoring_list[i]['Type'],
                        'Time': recoring_list[i]['Time'].tolist(),
                        'States': recoring_list[i]['States'].tolist(),
                        'NrDmps': recoring_list[i]['States'].shape[1]}

            dict_writer.writerow(new_data)


def compute_trajectory(data, i):
    weights = data['weights']
    way_points = data['way_points']

    nr_dmps = data['nr_dmps'][0]
    _ytrack = _dytrack = _ddytrack = None
```

```python

    dt = 0.01

    if nr_dmps == 3:
        _x = np.zeros((int(1.0 / dt * data['tau'][0]), 1))
        _y = np.zeros(_x.shape)
        _z = np.zeros(_x.shape)

        _dx = np.zeros(_x.shape)
        _dy = np.zeros(_x.shape)
        _dz = np.zeros(_x.shape)

        _ddx = np.zeros(_x.shape)
        _ddy = np.zeros(_x.shape)
        _ddz = np.zeros(_x.shape)

    _dmp = DmpDiscrete(nr_dmps=data['nr_dmps'][i], nr_bfs=data['nr_bfs'][i], tau=data['tau'][i])

    _y0 = way_points[:3, 0, i]
    _y0 = _y0[:, np.newaxis].T
    _dmp.y0 = _y0

    _g = way_points[:3, -1, i]
    _g = _g[:, np.newaxis].T
    _dmp.g = _g

    _dmp.w = weights[:, :, i]

    # roll out of the dmp
    _ytrack, _dytrack, _ddytrack = _dmp.roll_out()

    time = _dmp.cs.time

    # save the results for each dimension
    _x, _y, _z, _dx, _dy, _dz, _ddx, _ddy, _ddz = None, None, None, None, None, None, None, None, None
    if nr_dmps == 3:
        _x = _ytrack[:, 0]
        _y = _ytrack[:, 1]
        _z = _ytrack[:, 2]

        _dx = _dytrack[:, 0]
        _dy = _dytrack[:, 1]
        _dz = _dytrack[:, 2]

        _ddx = _ddytrack[:, 0]
        _ddy = _ddytrack[:, 1]
        _ddz = _ddytrack[:, 2]

```

```
    return _x, _y, _z, _dx, _dy, _dz, _ddx, _ddy, _ddz, time


def compute_mean_of_recording(recording_list):
    _max_length = len(recording_list[0]['States'][0, :])
    _time = None
    for i in range(len(recording_list)):
        if _max_length <= len(recording_list[i]['States']):
            _max_length = len(recording_list[i]['States'])
            _time = recording_list[i]['Time']

    _nan_array = np.empty((_max_length, len(recording_list)))
    _nan_array[:, :] = np.NaN

    if recording_list[0]['Type'] == "task":
        _x = _nan_array.copy()
        _y = _nan_array.copy()
        _z = _nan_array.copy()

        for i in range(len(recording_list)):
            _array_length = len(recording_list[i]['States'])
            _x[:_array_length, i] = recording_list[i]['States'][:, 0]
            _y[:_array_length, i] = recording_list[i]['States'][:, 1]
            _z[:_array_length, i] = recording_list[i]['States'][:, 2]

        _x_mean = np.nanmean(_x, axis=1)
        _y_mean = np.nanmean(_y, axis=1)
        _z_mean = np.nanmean(_z, axis=1)

        _trajectory_mean = np.vstack((_x_mean, _y_mean, _z_mean))

        return _trajectory_mean, _time


def perform_imitation(trajectory, nr_bfs=25, tau=1.0, y0=None, g=None,
    regression_type="RidgeRegression", imitation_type="eye"):
    # initialize DMPs
    _dmp = DmpDiscrete(nr_dmps=trajectory.shape[0], nr_bfs=nr_bfs, tau=
        tau, regression_type=regression_type, imitation_type=
        imitation_type)

    # set start and end
    for i in range(_dmp.nr_dmps):
        _dmp.y0[0, i] = y0[i]
        _dmp.g[0, i] = g[i]

    # Perform Imitation Learning
    _dmp.imitation_learning(trajectory.T)

    _ytrack, _dytrack, _ddytrack = _dmp.roll_out()
```

```
    return _dmp.cs.time, _ytrack.T, _dytrack.T, _ddytrack.T


def plot_rl(data):
    way_points = data['way_points']
    nr_dmps = data['nr_dmps'][0]

    _scaled_y, _scaled_dy, _scaled_ddy = None, None, None
    _non_scaled_y, _non_scaled_dy, _non_scaled_ddy = None, None, None

    dt = 0.01
    time = None
    _x, _y, _z, _dx, _dy, _dz, _ddx, _ddy, _ddz = None, None, None, None,
        None, None, None, None, None
    if nr_dmps == 3:
        _x = np.zeros((int(1.0 / dt * data['tau'][0]), data['nr_of_scaled
            '] + data['nr_of_non_scaled']))
        _y = np.zeros(_x.shape)
        _z = np.zeros(_x.shape)

        _dx = np.zeros(_x.shape)
        _dy = np.zeros(_x.shape)
        _dz = np.zeros(_x.shape)

        _ddx = np.zeros(_x.shape)
        _ddy = np.zeros(_x.shape)
        _ddz = np.zeros(_x.shape)

    for i in range(data['nr_of_scaled'] + data['nr_of_non_scaled']):
        _x[:, i], _y[:, i], _z[:, i], \
        _dx[:, i], _dy[:, i], _dz[:, i], \
        _ddx[:, i], _ddy[:, i], _ddz[:, i], time = compute_trajectory(
            data, i)

    if nr_dmps == 3:
        # for the scaled DMPs
        _x_mean_scaled = _x[:, :5].mean(axis=1)
        _y_mean_scaled = _y[:, :5].mean(axis=1)
        _z_mean_scaled = _z[:, :5].mean(axis=1)

        _x_std_scaled = _x[:, :5].std(axis=1)
        _y_std_scaled = _y[:, :5].std(axis=1)
        _z_std_scaled = _z[:, :5].std(axis=1)

        _x_confidence_scaled = 1.96 * _x_std_scaled / np.sqrt(data['
            nr_of_scaled'])
        _y_confidence_scaled = 1.96 * _x_std_scaled / np.sqrt(data['
            nr_of_scaled'])
        _z_confidence_scaled = 1.96 * _z_std_scaled / np.sqrt(data['
```

```
            nr_of_scaled'])

        _traj_scaled_mean = np.vstack((_x_mean_scaled, _y_mean_scaled,
            _z_mean_scaled))
        _traj_scaled_confidence = np.vstack((_x_confidence_scaled,
            _y_confidence_scaled, _z_confidence_scaled))

        # for the non scaled DMPs
        _x_mean_non_scaled = _x[:, 5:].mean(axis=1)
        _y_mean_non_scaled = _y[:, 5:].mean(axis=1)
        _z_mean_non_scaled = _z[:, 5:].mean(axis=1)

        _x_std_non_scaled = _x[:, :5].std(axis=1)
        _y_std_non_scaled = _y[:, :5].std(axis=1)
        _z_std_non_scaled = _z[:, :5].std(axis=1)

        _x_confidence_non_scaled = 1.96 * _x_std_non_scaled / np.sqrt(
            data['nr_of_non_scaled'])
        _y_confidence_non_scaled = 1.96 * _x_std_non_scaled / np.sqrt(
            data['nr_of_non_scaled'])
        _z_confidence_non_scaled = 1.96 * _z_std_non_scaled / np.sqrt(
            data['nr_of_non_scaled'])

        _traj_non_scaled_mean = np.vstack((_x_mean_non_scaled,
            _y_mean_non_scaled, _z_mean_non_scaled))
        _traj_non_scaled_confidence = np.vstack((_x_confidence_non_scaled
            , _y_confidence_non_scaled,
                                                 _z_confidence_non_scaled
                                                    ))

    # plot the scaled DMPs
    _fig_scaled, _axs_scaled = plt.subplots(nr_dmps, figsize=(10, 7))

    for i in range(nr_dmps):
        _axs_scaled[i].plot(time, _traj_scaled_mean[i, :], color='
            steelblue')
        _axs_scaled[i].fill_between(time, _traj_scaled_mean[i, :] +
            _traj_scaled_confidence[i, :],
                                    _traj_scaled_mean[i, :] -
                                        _traj_scaled_confidence[i, :],
                                        color='lightsteelblue')
        for j in range(data['nr_way_points'][0]):
            _axs_scaled[i].plot(way_points[-1, j], way_points[i, j], 'o')

    if nr_dmps == 3:
        _axs_scaled[0].set_ylabel('x-postion [m]', fontsize='large')
        _axs_scaled[1].set_ylabel('y-postion [m]', fontsize='large')
        _axs_scaled[2].set_ylabel('z-postion [m]', fontsize='large')
        _axs_scaled[2].set_xlabel('Time [s]', fontsize='large')
    _fig_scaled.tight_layout()
```

```
    plt.savefig('Plots/DMPs200Scale.png')

    # plot the non-scaled DMPs
    _fig_non_scaled, _axs_non_scaled = plt.subplots(nr_dmps, figsize=(10,
        7))

    for i in range(nr_dmps):
        _axs_non_scaled[i].plot(time, _traj_non_scaled_mean[i, :], color=
            'steelblue')
        _axs_non_scaled[i].fill_between(time, _traj_non_scaled_mean[i, :]
            + _traj_non_scaled_confidence[i, :],
                                        _traj_non_scaled_mean[i, :] -
                                            _traj_non_scaled_confidence[i,
                                            :],
                                        color='lightsteelblue')

        for j in range(data['nr_way_points'][0]):
            _axs_non_scaled[i].plot(way_points[-1, j], way_points[i, j],
                'o')

    if nr_dmps == 3:
        _axs_non_scaled[0].set_ylabel('x-postion [m]', fontsize='large')
        _axs_non_scaled[1].set_ylabel('y-postion [m]', fontsize='large')
        _axs_non_scaled[2].set_ylabel('z-postion [m]', fontsize='large')
        _axs_non_scaled[2].set_xlabel('Time [s]', fontsize='large')

    _fig_non_scaled.tight_layout()
    plt.savefig('Plots/DMPs200NonScale.png')

    return _traj_scaled_mean, _traj_non_scaled_mean, _axs_scaled,
        _axs_non_scaled


def plot_reward(data):
    rewards = data['rewards']

    _scaled = None
    _non_scaled = None
    for i in range(data['nr_of_scaled'] + data['nr_of_non_scaled']):
        if data['scaled'][i]:
            if _scaled is None:
                _scaled = data['rewards'][i]
            else:
                _scaled = np.vstack((_scaled, data['rewards'][i]))
        else:
            if _non_scaled is None:
                _non_scaled = data['rewards'][i]
            else:
                _non_scaled = np.vstack((_non_scaled, data['rewards'][i])
                    )
```

```python

    # generate a figure for scaled and non-scaled trajectory Learning
    nr_rewards = rewards[0].shape[0]
    episodes = np.arange(nr_rewards)

    if _scaled is not None:
        # compute the mean of each episode
        reward_mean = _scaled.mean(axis=0)
        reward_std = _scaled.std(axis=0)
        reward_confidence = 1.96 * reward_std / np.sqrt(data['
            nr_of_scaled'])

        fig_scaled, ax_scaled = plt.subplots(figsize=(10, 7))
        ax_scaled.plot(episodes, reward_mean, color='steelblue')
        ax_scaled.fill_between(episodes, reward_mean + reward_confidence,
             reward_mean - reward_confidence,
                                color='lightsteelblue')
        ax_scaled.set_yscale('log')
        ax_scaled.set_xlabel('Episodes', fontsize='large')
        ax_scaled.set_ylabel('Cost value', fontsize='large')
        ax_scaled.set_ylim([1e2, 5 * 1e5])
        plt.savefig('Plots/Reward200Scale.png')

    if _non_scaled is not None:
        # compute the mean of each episode
        reward_mean = _non_scaled.mean(axis=0)
        reward_std = _non_scaled.std(axis=0)
        reward_confidence = 1.96 * reward_std / np.sqrt(data['
            nr_of_non_scaled'])

        fig_scaled, ax_non_scaled = plt.subplots(figsize=(10, 7))
        ax_non_scaled.plot(episodes, reward_mean, color='steelblue')
        ax_non_scaled.fill_between(episodes, reward_mean +
            reward_confidence, reward_mean - reward_confidence,
                                    color='lightsteelblue')
        ax_non_scaled.set_yscale('log')
        ax_non_scaled.set_xlabel('Episodes', fontsize='large')
        ax_non_scaled.set_ylabel('Cost value', fontsize='large')
        ax_non_scaled.set_ylim([1e2, 5 * 1e5])
        plt.savefig('Plots/Reward200NonScale.png')


def plot_task_recordings(recording_list, used_trajectories, data):
    _fig_rec, _axs_rec = plt.subplots(3, figsize=(10, 7))

    # plot the trajectories of the simulated robot
    for i in range(len(recording_list)):
        trajectory = recording_list[i]['States']
        time = recording_list[i]['Time']

```

```
            for j in range(data['nr_dmps'][0]):
                _axs_rec[j].plot(time, trajectory[:, j])

        # plot the way points
        for i in used_trajectories:
            _way_points = data['way_points'][:, :, i]
            for j in range(data['nr_dmps'][0]):
                for k in range(1, data['nr_way_points'][0] - 1):
                    _axs_rec[j].plot(_way_points[-1, k], _way_points[j, k], '
                        x')

                _axs_rec[j].plot(_way_points[-1, 0], _way_points[j, 0], 'o')
                _axs_rec[j].plot(_way_points[-1, -1], _way_points[j, -1], 'o'
                    )

        fig, ax = plt.subplots()
        for i in range(len(recording_list)):
            time = recording_list[i]['Time']
            ax.plot(range(time.shape[0]), time)


    def plot_task_trajectory(data, used_trajectories):
        _fig_demo, _axs_demo = plt.subplots(3, figsize=(10, 7))

        for i in used_trajectories:
            _way_points = data['way_points'][:, :, i]

            for j in range(3):
                _x, _y, _z, _, _, _, _, _, _, time = compute_trajectory(data,
                    i)
                _trajectory = np.vstack((_x, _y, _z))
                _axs_demo[j].plot(time, _trajectory[j, :])

                for k in range(1, data['nr_way_points'][0] - 1):
                    _axs_demo[j].plot(_way_points[-1, k], _way_points[j, k],
                        'x')

                _axs_demo[j].plot(_way_points[-1, 0], _way_points[j, 0], 'o')
                _axs_demo[j].plot(_way_points[-1, -1], _way_points[j, -1], 'o
                    ')

        _axs_demo[0].set_ylabel('x-postion [m]', fontsize='large')
        _axs_demo[1].set_ylabel('y-postion [m]', fontsize='large')
        _axs_demo[2].set_ylabel('z-postion [m]', fontsize='large')
        _axs_demo[2].set_xlabel('Time [s]', fontsize='large')

        _fig_demo.tight_layout()


    def plot_imitation_learning(demonstration, demo_time, imitation,
```

```
    imitation_time):
    fig_imitation, axes_imitation = plt.subplots(demonstration.shape[0],
        figsize=(10, 7))

    plot_multi_trajectory(demonstration, demo_time, fig_imitation,
        axes_imitation)
    plot_multi_trajectory(imitation, imitation_time, fig_imitation,
        axes_imitation)

    if demonstration.shape[0] == 3:
        axes_imitation[0].set_ylabel('x-postion [m]', fontsize='large')
        axes_imitation[1].set_ylabel('y-postion [m]', fontsize='large')
        axes_imitation[2].set_ylabel('z-postion [m]', fontsize='large')
        axes_imitation[2].set_xlabel('Time [s]', fontsize='large')
    plt.savefig('Plots/ImitationRR.png')



def plot_multi_trajectory(trajectory, time, figure=None, axes=None):
    _fig = _axes = None

    if figure is None:
        _fig, _axes = plt.subplots(trajectory.shape[0], figsize=(10, 7))
    else:
        _fig = figure
        _axes = axes

    for i in range(trajectory.shape[0]):
        _axes[i].plot(time, trajectory[i, :])

    if trajectory.shape[0] == 3:
        _axes[0].set_ylabel('x-postion [m]', fontsize='large')
        _axes[1].set_ylabel('y-postion [m]', fontsize='large')
        _axes[2].set_ylabel('z-postion [m]', fontsize='large')
        _axes[2].set_xlabel('Time [s]', fontsize='large')

    _fig.tight_layout()


def plot_3d_trajectory(data, trajectory):
    way_points = data['way_points']

    fig3d = plt.figure()
    axes = plt.axes(projection='3d')

    # Trajectory plotting
    axes.plot3D(trajectory[0, :], trajectory[1, :], trajectory[2, :])
    # axes.view_init(60, 35)

    # plot the via-points
```

```
    for i in range(data['nr_way_points'][0]):
        axes.plot3D(way_points[0, i], way_points[1, i], way_points[2, i],
            'o')

    axes.set_xlabel('x (m)')
    axes.set_ylabel('y (m)')
    axes.set_zlabel('z (m)')

    plt.savefig('Plots/Trajectory3D.png')


def perform_trajectories(data, robot, used_trajectories, recording_type="
    task"):
    _recording_list = []

    # run the paths
    for i in used_trajectories:
        print("Trajectory number: {}".format(i))

        # compute the trajectory
        _x, _y, _z, _, _, _, _, _, _, time = compute_trajectory(data, i)
        _trajectory = np.vstack((_x, _y, _z))

        robot.reset_target_pose()

        robot.com.sim.startSimulation()
        print("Simulation started!")

        _recording_temp = robot.jacobian_inverse_control(_trajectory,
            recording_type=recording_type)

        robot.com.sim.stopSimulation()
        print("Simulation stopped!")
        _recording_list.append(_recording_temp)

    return _recording_list


def perform_trajectory(robot, trajectory):
    # reset target state
    robot.reset_target_pose()

    # start simulation
    robot.com.sim.startSimulation()
    print("Simulation started!")

    # traverse trajectory
    recording_temp = robot.jacobian_inverse_control(trajectory)

    # stop simulation
```

```
    robot.com.sim.stopSimulation()
    print("Simulation stopped!")


if __name__ == "__main__":
    data_rl = import_weight_data('Scaling/scaling.csv')
    plot_reward(data_rl)
    scaled, non_scaled, axes_scaled, axes_non_scaled = plot_rl(data_rl)
    # plot_3d_trajectory(data_rl, scaled)

    # Franka initialization
    print('Initialize Franka')
    panda = FrankaRobot(inverse_controller="CoppeliaSim")

    # Imitation learning

    data_im = import_weight_data('Imitation/weights.csv')
    used_traj = [0, 1, 2, 4, 8, 10, 12, 13, 14, 15, 16, 18]
    recordings = perform_trajectories(data_im, panda, used_traj)
    write_recordings(recordings)

    plot_task_trajectory(data_im, used_traj)
    rec_load = read_recordings('Imitation/recordings.csv')
    mean_trajectory, mean_time = compute_mean_of_recording(rec_load)
    perform_trajectory(panda, mean_trajectory)
    plot_task_recordings(rec_load, used_traj, data_im)

    # get start states and goal states from weights.csv for Imitation
       Learning
    start_y = data_im['way_points'][:3, 0, 0]
    goal_y = data_im['way_points'][:3, -1, 0]

    # Imitation Learning
    regression_type = "RidgeRegression"
    imitation_type = "eye"
    imitation_t, imitation_y, _, _ = perform_imitation(mean_trajectory,
                                                       nr_bfs=25,
                                                       tau=9.5,
                                                       y0=start_y,
                                                       g=goal_y,
                                                       regression_type=
                                                          regression_type
                                                          ,
                                                       imitation_type=
                                                          imitation_type)

    plot_imitation_learning(mean_trajectory, mean_time, imitation_y,
       imitation_t)

    perform_trajectory(panda, imitation_y)
```

```
    plot_3d_trajectory(data_rl, imitation_y)

    plt.show()
```

B APPENDIX TWO

B.1 Abbreviations

C-Space - Configuration Space
CMA-ES - Covariance Matrix Adaption Evolution Strategy
DH - Denavit-Hartenberg
DMP - Dynamic Movement Primitives
DOF - degree of freedom
IM - Imitation Learning
MDP - Markov Decision Process
ProMP - Probabilistic Movement Primitives
RL - Reinforcement Learning
ROS - Robot Operating System

B.2 Symbols

The usual vector notation was used, where the scalars are lower case letters, a, row vectors are written in bold, $\mathbf{v}$, and matrices are written in bold and capital letters, $\mathbf{M}$. A list of all symbols is given arranged according to the sections of the thesis.

B.2.1 Robot Basics

X, Y, Z, coordinate of the position
θ, ϕ, ψ, coordinate of the orientation
N, number of ridgid bodies
m, number of the DOFs of the ridgid bodies
J, number of Joints
f_i, DOFs of the corresponding joint i

B.2.2 Kinematics and Dynamics

$\hat{x}, \hat{y}, \hat{z}$, transformed coordinates
x, y, z, input coodrinates
θ, ϕ, ψ, angles of rotation referred to the x, y, z-axis
a, b, c, displacements along the x, y, z- axis
$\mathbf{q}$, configuration of a robot
$\mathbf{x}$, state of the end-effector $(x, y, z, \theta, \phi, \psi)$
$\mathbf{T}_{i-1,i}$, Transformation Matrix
θ_i, DH-Parameter: joint angle
d_i, DH-Parameter: link offset
a_i, DH-Parameter: link length
α_i, DH-Parameter: link twist

B.2.3 Robot Control

$\mathbf{J}$, Jacobian Matrix
$\delta\mathbf{x}$, change of the Cartisian position of the end-effector

$\delta\mathbf{q}$, change of the joint angles of the robot
η, step size of the Inverse Controller
τ, torque of the revolute joints
$\mathcal{F}$, force vector

B.2.4 Reinforcement Learning (RL)

a_t, action at time t
s_t, state at time t
f_π, deterministic policy
$\pi(a_t|s_t)$, stochastic policy for the given state s_t to perform the action a_t
$R(\tau)$, return of the trajectory τ
r_t, reward at time t

B.2.5 CMA-ES

n, search space dimension
g, generation counter $\lambda \geq 2$, population size, sample size
w_i', weight helper parameter
$\mu < \lambda$, number of positively selected search point in the population
μ_{eff}, the variance effective selection mass of the mean
$c_\sigma < 1$, learning rate for the cumulation for the step-size control
$d_\sigma \approx 0$, damping parameter for the step-size update
$c_c \leq 1$, learning rate for cumulation for the rank-one update of the covariance matrix
$c_1 \leq 1 - c_\mu$, learning rate for the rank-one update of the covariance matrix update
c_μ, learning rate for the rank-μ update of the covariance matrix update
$\alpha_\mu^-, \alpha_{\mu_{eff}}^-, \alpha_{posdef}^-$, helper parameter
w_i, weight of the CMA-ES Algorithm
$\sigma^{(g)} > 0$, step-size
$\mathbf{B} \in \mathbb{R}^n$, an orthogonal matrix. Columns of $mathbfB$ are eigenvectors of $\mathbf{C}$ with unit length and correspond to the diagonal elements of $\mathbf{D}$
$\mathbf{C}^{(g)} \in \mathbb{R}^{n\times n}$, covariance matrix at generation g
$\mathbf{D} \in \mathbb{R}^{n\times n}$ diagonal matrix with the squared eigenvalues of $\mathbf{C}$
$\mathbf{m}^{(g)} \in \mathbb{R}^n$, mean value of the search distribution at generation g
$\mathbf{x}_i \in \mathbb{R}^n$, i-th sample of the multivariant normal distribution
$\mathbf{p} \in \mathbb{R}^n$, evolution path, a sequence of successive (normalized) steps, the strategy takes over a number of generations.

B.2.6 Dynamic Movement Primitive

τ, temporal scaling factor
α_z, β_z, spring and damping coeffitients
g, goal state
y_0, initial state
y, position
$\dot{y}$, velocity of y
$\ddot{y}$, acceloration of y
f, forcing term

$\mathbf{\Psi}_i$, i-th base function
w_i, i-th weight
c_i, i-th mean value of the base function
σ_i, i-th standard deviation of the base function
x, variable of the canonical system for a discrete DMP
ϕ, variable of the canonical system for a harmonic DMP
a_x, decay parameter of the canonical system
$\mathbf{\Gamma}$, third derivatieve of the base functions
λ, penalty term of the regression
$C(\tau)$, cost function for the trajectory τ of the RL Algorithm
$\mathbf{R}$, penalty matrizes of the cost function $C(\tau)$

B.2.7 Experiments

d, Number of Dynamic Movement Primitives
b, Number of basis functions
$\mathbf{x}_i \in \mathbb{R}^{db\times 1}$, non-scaled weight vector of the DMPs
$\mathbf{c}_{scale} \in \mathbb{R}^{db\times 1}$, scaling vector

C APPENDIX THREE

C.1 CMA-ES Parameters

In the 4 the parameters of the CMA-ES algorithm are described and their initialization is given.

C.2 Robot Specifications

In the tables given here, the Cartesian and joint limits of the Franka Emika Panda are given.

C.2.1 Cartesian Limits

C.2.2 Joint Limits

Symbol	Name	Equation
n	search space dimension	
λ	population size, sample size	$\lambda = 4 + \lfloor 3\ln(n) \rfloor$
w'_i	weight helper parameter	$w'_i = \ln\left(\frac{\lambda+1}{2}\right) - \ln(i)$, for $i = 1; \ldots, \lambda$
μ	number of (positively) selected search points in the population	$\mu = \lvert\{w_i > 0\}\rvert = \lfloor \lambda/2 \rfloor$
μ_{eff}	variance effective selection mass for the mean	$\mu_{eff} = \frac{(\sum_{i=1}^{\mu} w'_i)^2}{\sum_{i=1}^{\mu} w'^2_i} \in [1, \mu]$
μ^-_{eff}	negative μ_{eff}	$\mu^-_{eff} = \frac{(\sum_{i=\mu+1}^{\lambda} w'_i)^2}{\sum_{i=\mu+1}^{\lambda} w'^2_i} \in [1, \mu]$
c_σ	learning rate for the cumulation for the step-size control	$c_\sigma = \frac{\mu_{eff}+2}{n+\mu_{eff}+5}$
d_σ	damping parameter for step size update	$d_\sigma = 1 + 2\max(0, \sqrt{\frac{\mu_{eff}-1}{n+1}}) + c_\sigma$
c_c	learning rate for the cumulation for the rank-one update of the covariance matrix	$c_c = \frac{4+\mu_{eff}/n}{n+4+2\ \mu_{eff}/n}$
c_1	learning rate for the rank-one update of the covariance matrix update	$c_1 = \frac{\alpha_{cov}}{(n+1.3)^2+\mu_{eff}}$ with $\alpha_{cov} = 2$
c_μ	learning rate for the rank-µ update of the covariance matrix update	$c_\mu = \min(1 - c_1, \alpha_{cov}\frac{\mu_{eff}-2+1/\mu_{eff}}{(n+2)^2+\alpha_{cov}\mu_{eff}/2})$ with $\alpha_{cov} = 2$
α^-_μ	helper parameter	$\alpha^-_\mu = 1 + c_1/c_\mu$
$\alpha^-_{\mu_{eff}}$	helper parameter	$\alpha_{\mu_{eff}} = 1 + \frac{2\mu^-_{eff}}{\mu_{eff}+2}$
α^-_{posdef}	helper parameter	$\alpha^-_{posdef} = \frac{1-c_1-c_\mu}{nc_\mu}$
w_i	weights	$w_i = \begin{cases} \frac{1}{\sum \lvert w'_j\rvert^+} w'_i & \text{if } w'_i \geq 0 \\ \frac{\min(\alpha^-_\mu, \alpha^-_{\mu_{eff}}, \alpha^-_{posdef})}{\sum \lvert w'_j\rvert^-} w'_i & \text{if } w'_i < 0 \end{cases}$

Table 4: *This table lists the CMA-ES parameters.*

Name	Translation	Rotation	Elbow
$\dot{p}_{max}$	$1.7\frac{m}{s}$	$2.5\frac{rad}{s}$	$2.175\frac{rad}{s}$
$\ddot{p}_{max}$	$13.0\frac{m}{s^2}$	$25.0\frac{rad}{s^2}$	$10.0\frac{rad}{s^2}$
$\dddot{p}_{max}$	$6500.0\frac{m}{s^3}$	$12500.0\frac{rad}{s^3}$	$5000.0\frac{rad}{s^3}$

Table 5: *Cartesian Limits of the Franka Emika Panda*

Name	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6	Joint 7	Unit
q_{max}	2.8973	1.7628	2.8973	-0.0698	2.8973	3.7525	2.8973	rad
q_{min}	-2.8973	-1.7628	-2.8973	-3.0718	-2.8973	-0.0175	-2.8973	rad
$\dot{q}_{max}$	2.175	2.175	2.175	2.175	2.610	2.610	2.610	$\frac{rad}{s}$
$\ddot{q}_max$	15	7.5	10	12.5	15	20	20	$\frac{rad}{s^2}$
$\dddot{q}_max$	7500	3750	5000	6250	7500	10000	10000	$\frac{rad}{s^3}$
τ_{jmax}	87	87	87	87	12	12	12	Nm
$\dot{\tau}_{jmax}$	1000	1000	1000	1000	1000	1000	1000	$\frac{Nm}{s}$

Table 6: *Cartesian Limits of the Franka Emika Panda*