TRAJECTORY GENERATION FOR A MULTIBODY ROBOTIC SYSTEM: MODERN METHODS BASED ON PRODUCT OF EXPONENTIALS

By

Aryslan Malik

A Dissertation Submitted to the Faculty of Embry-Riddle Aeronautical University

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Aerospace Engineering

December 2021

Embry-Riddle Aeronautical University

Daytona Beach, Florida

TRAJECTORY GENERATION FOR A MULTIBODY ROBOTIC SYSTEM: MODERNMETHODS BASED ON PRODUCT OF EXPONENTIALS

By

Aryslan Malik

This Dissertation was prepared under the direction of the candidate’s Dissertation Committee Chair, Dr. Troy Henderson, Department of Aerospace Engineering, and has been approved by the members of the Dissertation Committee. It was submitted to the Office of the Senior Vice President for Academic Affairs and Provost, and was accepted in the partial fulfillment of the requirements for the Degree of Philosophy in Aerospace Engineering.

DISSERTATION COMMITTEE

Troy A. Henderson Digitally signed by Troy A. Henderson Date: 2021.12.08 15:56:40 -05'00'	Morad Nazari Digitally signed by Morad Nazari Date: 2021.12.09 09:39:57 -05'00'
Chairman, Dr. Troy Henderson	Member, Dr. Morad Nazari
Richard J. Prazenica Digitally signed by Richard J. Prazenica DN: cn=Richard J. Prazenica, o, ou=ERAU, email=prazenir@erau.edu, c=US Date: 2021.12.08 15:59:13 -05'00'	William MacKunis Digitally signed by William MacKunis Date: 2021.12.09 10:55:06 -05'00'
Chairman, Dr. Richard Prazenica	Member, Dr. William MacKunis
Sirish Namilae Digitally signed by Sirish Namilae Date: 2021.12.09 11:27:41 -05'00'	
Graduate Program Coordinator, Dr. Sirish Namilae	Date
James W. Gregory Digitally signed by James W. Gregory DN: cn=James W. Gregory, o=Embry Riddle Aeronautical Unversity, ou=College of Engineering, email=gregoj14@erau.edu, c=US Date: 2021.12.09 13:10:48 -05'00'	
Dean of the College of Engineering, Dr. James W. Gregory	Date
Lon Moeller Digitally signed by Lon Moeller Date: 2021.12.09 13:28:07 -05'00'	
Senior Vice President for Academic Affairs and Provost, Dr. Lon Moeller	Date

ACKNOWLEDGEMENTS

First of all, I am grateful for all the given opportunities and the real treasure, which I view as great and wonderful people that I met on this journey.

It is a genuine pleasure to express my deep sense of gratitude to my advisors, mentors, and friends, Dr. Troy Henderson and Dr. Richard Prazenica for giving me the greatest opportunity in the time when I was at a crossroads. Thank you for your patient support and guidance, your trust, and freedom you gave me to pursue research that I deemed to be the most interesting and exciting.

I would like to thank my dissertation committee members, Dr. Morad Nazari and Dr. William MacKunis, for their insightful comments, challenging questions, and encouragement.

Finishing this work without any help would have been an insurmountable task. Thank you, Daniel Posada, Yevgeniy Lischuk, Stanislav Sikulskyi, Chris Hays, everyone at the Space Technologies Laboratory, my friends for your genuine involvement, academic and mental support in the adverse times and most importantly coffee breaks.

My sincere thanks also goes to everyone in ERAU Aerospace Engineering Department for giving me this opportunity to pursue a PhD degree, special thanks to Dr. Tasos Lyrintzis whom I thank profusely for constant administrative support throughout my studies.

Finally, I want to express my deep sense of thanks to my family. My parents, I am forever grateful for your continued support and encouragement. My wife, for always being there for me during hardships and happy moments alike. Without you, I doubt that this endeavor would have been possible.

ABSTRACT

This work presents several trajectory generation algorithms for multibody robotic systems based on the Product of Exponentials (PoE) formulation, also known as screw theory. A PoE formulation is first developed to model the kinematics and dynamics of a multibody robotic manipulator (Sawyer Robot) with 7 revolute joints and an end-effector. In the first method, an Inverse Kinematics (IK) algorithm based on the Newton-Raphson iterative method is applied to generate constrained joint-space trajectories corresponding to straight-line and curvilinear motions of the end effector in Cartesian space with finite jerk. The second approach describes Constant Screw Axis (CSA) trajectories which are generated using Machine Learning (ML) and Artificial Neural Networks (ANNs) techniques. The CSA method smooths the trajectory in the Special Euclidean (SE(3)) space. In the third approach, a multi-objective Swarm Intelligence (SI) trajectory generation algorithm is developed, where the IK problem is tackled using a combined SI-PoE ML technique resulting in a joint trajectory that avoids obstacles in the workspace, and satisfies the finite jerk constraint on end-effector while minimizing the torque profiles. The final method is a different approach to solving the IK problem using the Deep Q-Learning (DQN) Reinforcement Learning (RL) algorithm which can generate different joint space trajectories given the Cartesian end-effector path.

For all methods above, the Newton-Euler recursive algorithm is implemented to compute the inverse dynamics, which generates the joint torques profiles. The simulated torque profiles are experimentally validated by feeding the generated joint trajectories to the Sawyer robotic arm through the developed Robot Operating System (ROS) - Python environment in the Software Development Kit (SDK) mode. The developed algorithms can be used to generate various trajectories for robotic arms (e.g. spacecraft servicing missions).

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

Figure Page

LIST OF TABLES

ABBREVIATIONS

PoE	Product of Exponentials
DoF	Degree of Freedom
FK	Forward Kinematics
IK	Inverse Kinematics
D-H	Denavit-Hartenberg
ML	Machine Learning
RL	Reinforcement Learning
DQN	Deep Q-Learning
DDPG	Deep Deterministic Policy Gradient
NAF	Normalized Advantage Function
ANN	Artificial Neural Network
AI	Artificial Intelligence
SI	Swarm Intelligence
PSO	Particle Swarm Optimization
CTC	Computed Torque Controller
SE(3)	Special Euclidean Space

1. Introduction

This chapter explains the motivation behind the research done on joint space trajectory generation for high DoF robotic systems. It outlines the problem statement and specific objectives related to the generation of joint space trajectories given a Cartesian end-effector trajectory or path. Also, it provides background information on forward kinematics, inverse kinematics and dynamics, the mathematical foundation, and robotic arm that was used in implementation phase of the research work.

1.1. Problem Statement and Objectives

Since the 1980's, a popular approach for studying and analyzing the dynamics and kinematics of open-chain mechanisms and multibody systems has been the use of Denavit-Hartenberg (D-H) parameters. However, recently, there has been a rise of interest in the PoE formulation used for forward and velocity kinematics of multibody systems. Yet, there are few works that provide extensive and exhaustive description, simulation and implementation focused on developing modern trajectory generation algorithms for robotic manipulators using the Product of Exponentials formulation as a Forward Kinematics tool. This work describes the application of Machine Learning (ML), Artificial Neural Networks (ANN), Artificial Intelligence (AI), and Reinforcement Learning (RL) in solving the Inverse Kinematics (IK) problem for high Degree of Freedom (DoF) robotic arms. Depending on the number of joints and links possessed by a manipulator, the mapping from Cartesian space to the robot's joint space becomes very involved, which is problematic as the tasks that a robotic arm receives are in Cartesian space, whereas the commands (velocity or torque) are in joint space. Thus, the IK solution is an important step for achieving "full" autonomy of robotic systems, which can be approached by using techniques inspired by brain activity that controls several open-chain manipulators at once, instantaneously satisfying multiple objectives. Although most of the ML and AI techniques are mature when it comes to vision systems, the algorithms and architectures of RL models have to be optimized for robotic systems to achieve better performance. This research extends the utility of the learning algorithms to the robotic arm manipulation framework.

The aim of this research is to consolidate the algorithms and formulations of forward/inverse kinematics and dynamics, Bézier curves, ML, and RL, and to present simulation case studies and experimental implementations to demonstrate this unified approach. The research work aims to accomplish this goal by presenting several trajectory generation algorithms based on the Product of Exponentials (PoE) formulation, also known as screw theory, and apply these algorithms to Rethink Robotics' "Sawyer" robotic arm (Robotics, 2017). This research also experimentally compares the developed PoE Forward Kinematics model with the "classical" D-H parameters approach in terms of joint angle errors and joint torques of a "Sawyer" multibody robotic system.

1.2. Forward Kinematics, Inverse Kinematics and Dynamics

Aerospace robotic systems, such as robotic arms performing spacecraft servicing missions, require precise, real-time trajectory planning and control. Since the 1980's, a popular approach for studying and analyzing the dynamics and kinematics of open-chain mechanisms and multibody systems has been the use of D-H parameters (Lynch & Park, 2017). Although the D-H formulation requires a minimum number of parameters to describe a robot's kinematics, it follows a strict convention as to how the reference frames are assigned to each link. The PoE formulation, based on Lie group theory, Lie algebra, and screw theory, on the other hand, does not impose such restrictions at a cost of not utilizing a minimal set of parameters required to describe the kinematics of a robot.

The PoE formulation of forward kinematics was first described by Brockett in (Brockett, 1984), and can also be found in (Lynch & Park, 2017; Murray et al., 2017; F. C. Park, 1994; Selig, 2004a,b). This work was partly motivated by the recent interest in the PoE formulation. In (C. He et al., 2013; Wang et al., 2018) the PoE formulation is applied to carry out kinematic analysis of robotic manipulators. In (Chirikjian, 2018; Müller, 2019; F. C. Park et al., 2018), a general overview of the PoE method is provided with examples and a tutorial. Lastly, in (G. Chen et al., 2014; R. He et al., 2010; C. Li et al., 2016; Pac & Popa, 2013) the authors address kinematic errors and calibration of manipulators using PoE. The PoE can also be used to define orbits and the state of a satellite (Malik, Henderson, & Prazenica, 2021c).

A conventional approach for solving the inverse kinematics problem involves using the Newton-Raphson iterative method of nonlinear root finding. Examples of the usage of this method are shown in (Chiaverini et al., 2016; Lynch & Park, 2017; Murray et al., 2017; Selig, 2004a). Alternatively, a “lookup” method of inverse kinematics approximation can be used (Tarokh & Kim, 2007). In this work, the conventional method is developed and described in Chapter 3, which is followed by modern trajectory generation algorithms that are outlined in Chapters 4, 5, and 6. The feedforward plus feedback linearizing controller used in this work requires the computation of the inverse dynamics, an algorithm outlined in (Featherstone, 1983, 2014; Lynch & Park, 2017; Murray et al., 2017; Selig, 2004a).

1.3. Mathematical Background

Lie algebra and Lie Group theory can be used in the formulation of forward and inverse kinematics as well as dynamics (Lynch & Park, 2017). The equations of motion of an open-chain robot with n single degree of freedom revolute joints can be expressed as a set of second-order differential equations in the general form:

$$\tau = M(\theta)\ddot{\theta} + h(\theta, \dot{\theta}) \tag{1.1}$$

where $\theta \in \mathbb{R}^n$ is the vector of joint variables, $\tau \in \mathbb{R}^n$ is the vector of joint torques, $M(\theta) \in \mathbb{R}^{n\times n}$ is the symmetric positive-definite configuration-dependent mass matrix, and $h(\theta, \dot{\theta}) \in \mathbb{R}^n$ is the combination of centripetal, Coriolis, gravity, and friction terms applied to the system that depend on θ and $\dot{\theta}$. Equation 1.1 calculates the joint torques and requires the prior knowledge of a robot's state $(\theta, \dot{\theta})$ and desired acceleration $\ddot{\theta}$; thus, it is referred to as inverse dynamics. The forward dynamics problem, on the other hand, is concerned with finding a robot's acceleration $\ddot{\theta}$ given its state $(\theta, \dot{\theta})$ and the joint torques:

$$\ddot{\theta} = M^{-1}(\theta)(\tau - h(\theta, \dot{\theta})) \tag{1.2}$$

In the case where an external force is applied at the tip of the end-effector, the formulation of the forward dynamics is as follows:

$$M(\theta)\ddot{\theta} = \tau - h(\theta, \dot{\theta}) - J^T(\theta)\mathcal{F}_{\text{tip}} \tag{1.3}$$

where $\mathcal{F}_{\text{tip}}$ is the wrench applied by the end-effector and $J(\theta)$ is the manipulator Jacobian matrix that linearly relates the tip velocity v_{tip} to the joint velocity vector $\dot{\theta}$.

$$v_{\text{tip}} = J(\theta)\dot{\theta} \tag{1.4}$$

The wrench $\mathcal{F}_{\text{tip}} \in \mathbb{R}^6$ shown in Equation 1.3 is described below. By defining an arbitrary reference frame $\{a\}$, let $r_a \in \mathbb{R}^3$ be the vector to the point where force $f_a \in \mathbb{R}^3$

is applied. Thus, the wrench $\mathcal{F}_a$ expressed in the $\{a\}$ frame is as follows:

$$\mathcal{F}_a = \begin{bmatrix} m_a \\ f_a \end{bmatrix} = \begin{bmatrix} r_a \times f_a \\ f_a \end{bmatrix} = \begin{bmatrix} [r_a] f_a \\ f_a \end{bmatrix} \in \mathbb{R}^6 \tag{1.5}$$

where $[r_a] \in so(3)$ is a 3×3 skew-symmetric matrix representation of $r_a \in \mathbb{R}^3$.

The position and orientation of the end-effector or any other frame is represented as a 4×4 transformation matrix $T \in SE(3)$ which has the following formulation:

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} \tag{1.6}$$

where $R \in SO(3)$ is the orientation of the body represented as a member of the special orthogonal group and $p \in \mathbb{R}^3$ is a vector that represents the position of the frame in Cartesian space. The transformation matrix can be obtained using the products of exponentials formula that can be used for any open-chain manipulator (Brockett, 1984; Lynch & Park, 2017; F. C. Park, 1994). The formulation is shown below, where the position and orientation of the end-effector or any other frame can be represented in the space (inertial) frame, given screw axes in space (inertial) or body frame, and the home configuration of that frame $M_{sn} \in SE(3)$ in the space frame:

$$T_{sn} = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \cdots e^{[\mathcal{S}_n]\theta_n} M_{sn} \tag{1.7}$$

$$T_{sn} = M_{sn} e^{[\mathcal{B}_1]\theta_1} e^{[\mathcal{B}_2]\theta_2} \cdots e^{[\mathcal{B}_n]\theta_n} \tag{1.8}$$

and, for every $i = 1, 2, \cdots, n$, the norm of angular velocity $\|\omega_i\| = 1$ and v_i is the linear velocity at the inertial frame's origin, expressed in the inertial frame produced purely due to the rotation about the screw axis $(v_i = -\omega_i \times r_q)$, where q is a point on the screw axis.

In the equation above unit screw axes in the body frame, i.e. $\mathcal{B}_i \ (i = 1, 2, \cdots, n)$, can be obtained in terms of unit screw axes in the inertial frame, $\mathcal{S}_i \ (i = 1, 2, \cdots, n)$, using the adjoint map:

$$\mathcal{B}_i = [\mathrm{Ad}_{M_{sn}^{-1}}]\mathcal{S}_i = \begin{bmatrix} R_i & 0 \\ [p_i]R_i & R_i \end{bmatrix} \mathcal{S}_i \tag{1.9}$$

where,

$$\mathcal{S}_i = \begin{bmatrix} \omega_i \\ v_i \end{bmatrix} \in \mathbb{R}^6 \tag{1.10}$$

Also, the rows of the “Space Jacobian” matrix shown in Equation 1.4 are constructed from screw axes, such that:

$$J_{si}(\theta) = \mathrm{Ad}_{e^{[\mathcal{S}_1]\theta_1} \cdots e^{[\mathcal{S}_{i-1}]\theta_{i-1}}}(\mathcal{S}_i) \tag{1.11}$$

The above formulation requires the “home” configuration $M_{sn} \in SE(3)$ of the frame in question relative to the fixed base (inertial) frame when the robot is in its zero position (i.e. joint variables $\theta_1, \theta_2, \cdots, \theta_n$ are zero). Square brackets represent skew-symmetric mapping of an element such that: $p \in \mathbb{R}^3 \to [p] \in so(3)$ and $\mathcal{S} \in \mathbb{R}^6 \to [\mathcal{S}] \in se(3)$. For instance,

$$[\mathcal{S}] = \begin{bmatrix} [\omega] & v \\ 0 & 0 \end{bmatrix} \tag{1.12}$$

Forward kinematics utilizing the products of exponentials formulation utilizes the exponential mapping such that: $[\mathcal{S}]\theta \in se(3) \to T \in SE(3)$.

1.4. Robotic Arm Description

The 7R manipulator considered in this work is Rethink Robotics' Sawyer robot. The robot has 7 links and 7 joints, out of which 4 are rolling and 3 are pitching joints, as shown in Figure 1.1 (Robotics, 2017). The masses of each link and the position of

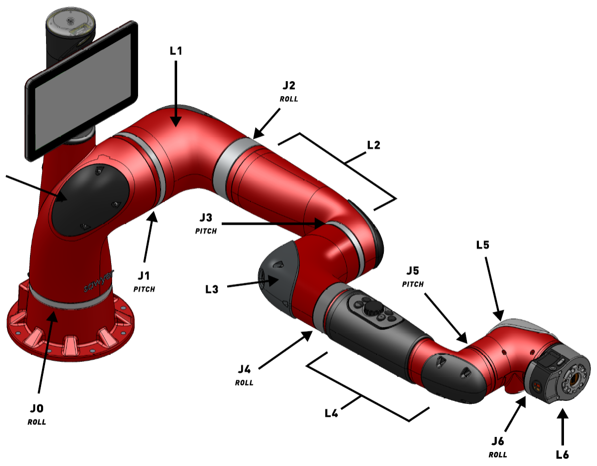

Figure 1.1 Sawyer joints and links (Robotics, 2017)

each link's center of gravity were recorded from the Universal Robot Description Format (URDF) file dedicated to the Sawyer robot (Robotics, 2017). The mass of each link and the position of its center of gravity represented in the inertial/space (base) frame (x_s, y_s, z_s) are given in Table 1.1. In addition, the inertia matrix of each link was obtained

Table 1.1

Link masses and centers of gravity

Link	Mass (kg)	x_s (m)	y_s (m)	z_s (m)
1	5.32	0.0244	0.0110	0.2236
2	4.51	0.1078	0.1425	0.3201
3	1.75	0.03568	0.1775	0.3172
4	2.51	0.5091	0.0663	0.3218
5	1.12	0.7301	0.0309	0.3189
6	1.56	0.9047	0.1314	0.3109
7	0.33	0.9860	0.1517	0.3170

from the URDF file. It is important to mention that inertia matrices were given in each link's frame that is attached to its center of gravity with a specific orientation that follows URDF guidelines. Table 1.2 shows the inertia components of each link as given in the

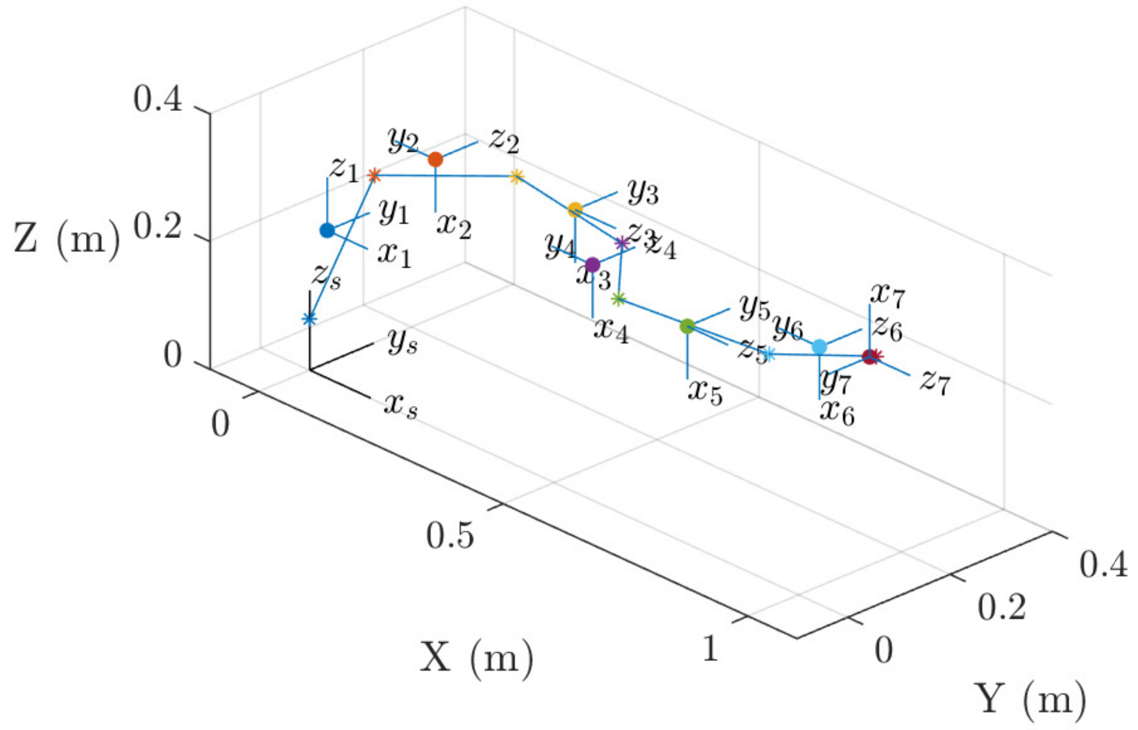

Figure 1.2 Home configuration of the Sawyer robot

URDF file and represented in the body frame of each link, as shown in Figure 1.2, where the link body frames are attached to the center of gravity of each link with the orientation as described in the URDF file.

Solid circles represent centers of gravity and stars are physical locations of joints. The physical locations of joints are crucial for the screw axes representation. Screw axes in the space frame are given below:

$$\mathcal{S}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \mathcal{S}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ -0.317 \\ 0 \\ 0.081 \end{bmatrix} \mathcal{S}_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0.317 \\ -0.1925 \end{bmatrix} \mathcal{S}_4 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ -0.317 \\ 0 \\ 0.481 \end{bmatrix} \tag{1.13}$$

$$\mathcal{S}_5 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0.317 \\ -0.024 \end{bmatrix} \mathcal{S}_6 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ -0.317 \\ 0 \\ 0.881 \end{bmatrix} \mathcal{S}_7 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0.317 \\ -0.1603 \end{bmatrix} \tag{1.14}$$

The end-effector home configuration in the space (inertial) frame is given as:

$$M_{s7} = \begin{bmatrix} 0 & 0 & 1 & 0.9860 \\ 0 & -1 & 0 & 0.1517 \\ 1 & 0 & 0 & 0.3170 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.15}$$

Similarly, home configurations in $SE(3)$ were obtained for centers of gravity of each link.

Table 1.2

Link inertia properties in gm^2

Link	I_{xx}	I_{xy}	I_{xz}	I_{yy}	I_{yz}	I_{zz}
1	53.31	4.71	11.73	57.90	8.02	23.66
2	22.40	-0.24	-0.29	14.61	-6.09	17.30
3	25.51	0.00	0.01	25.30	-3.32	3.42
4	10.16	-0.01	0.27	6.57	3.03	6.91
5	13.56	0.02	0.14	13.56	1.06	1.37
6	4.73	0.12	0.05	2.97	-1.16	3.18
7	0.31	0.00	0.00	0.22	-0.01	0.36

1.5. Contributions

This work consolidates the algorithms and formulations of forward/inverse kinematics and dynamics and presents several modern trajectory generation algorithms in simulation case studies and experimental implementation. The general overview of the work is presented as follows.

A PoE formulation is first developed to model the kinematics and dynamics of a multibody robotic manipulator with 7 revolute joints and an end effector. Using the developed model, an open-loop simulation is carried out. The simulation results are compared to the classical D-H parameters approach and experimental results using a simplistic torque trajectory applied to the "Sawyer" robotic arm.

Subsequently, two closed-loop simulations are presented. The first closed-loop simulation tracks a straight line trajectory and the second closed-loop simulation makes the end-effector track a more complex curvilinear trajectory produced from Bézier curves under specific constraints. Both of these simulations follow similar steps as explained below. An inverse kinematic algorithm based on the Newton-Raphson iterative method is applied to generate constrained joint-space trajectories corresponding to straight-line (first closed-loop simulation) and curvilinear (second closed-loop simulation) motion of the end effector in Cartesian space with finite jerk. Derivatives of these joint-space trajectories are computed using Bézier curves, which ensures dynamically feasible trajectories. A novel method of Mean Arctangent Absolute Percentage Error (MAAPE) is then used to check accuracy of the derivatives of joint-space trajectories. The Newton-Euler recursive algorithm is then implemented to compute the inverse dynamics, which generates the joint torques required to achieve the reference trajectories. These torques are then incorporated into a closed-loop control algorithm, and simulation studies are performed using a dynamic model of the robotic system.

It is demonstrated that the proposed approach is able to successfully generate constrained trajectories, and by using MAAPE it is shown that lower order Bézier curves approximate the joint-space derivatives with the same accuracy as higher order polynomials. The performance of the closed-loop controller in terms of accuracy of reference trajectory tracking subject to model uncertainties was checked. The significant contribution of this method is integration of all of the aforementioned techniques applied to a robotic system. Moreover, it is believed that this algorithm will be able to generate trajectories for robotic arms performing spacecraft servicing missions, because it also takes into account the variable gravity vector.

The next algorithm generates trajectories with constant screw axes, which smooths the trajectories in Special Euclidean space. These trajectories' paths are defined by Bézier curves. The time-scaling is quintic polynomial, which ensures that the jerk is finite throughout the trajectory. The CSA trajectories are obtained by varying multiple parameters that are chosen by Particle Swarm Optimization (PSO), Q-Learning, and DQN algorithms. The performances of these three approaches are compared.

Multi-objective swarm intelligence trajectory generation (SI-PoE) is the next algorithm described in this work. Given a priori knowledge of the end-effector Cartesian trajectory and obstacles in the workspace, the inverse kinematics problem is tackled by the SI-PoE algorithm subject to multiple constraints. The algorithm is designed to satisfy a finite jerk constraint on the end-effector, avoid obstacles, and minimize control effort while tracking the Cartesian trajectory. The SI-POE algorithm is compared with conventional inverse kinematics algorithms and standard PSO. The joint trajectories produced by SI-POE are experimentally tested on the Sawyer 7 DoF robotic arm, and the resulting torque trajectories are compared.

The final algorithm considered is DQN-IK, which is a deep reinforcement learning approach for the inverse kinematics solution of a high degree of freedom manipulator. A Deep Reinforcement Learning (RL) approach for solving the Inverse Kinematics (IK) problem of a 7-Degree of Freedom (DoF) robotic manipulator using Product of Exponentials (PoE) as a Forward Kinematics (FK) computation tool and the Deep Q-Network (DQN) as an IK solver is developed. The algorithm is designed to produce joint space trajectories from a given end-effector trajectory. Different network architectures were explored and the output of the DQN was implemented on the Sawyer robotic arm. The DQN was able to find different trajectories corresponding to a single Cartesian path of the end-effector. The network agent was able to learn random Bézier end-effector trajectories in a reasonable time frame with good accuracy, demonstrating that even though DQN is mainly used in discrete solution spaces, it could also be applied for generating joint space trajectories.

This document is organized as follows: Chapter 1 gives insight into the problem statement, the objectives, the challenges, the proposed solution and the cases for testing. Chapter 2 will provide a description of the state-of-the-art and a literature review. Chapter 3 provides an overview of the Newton-Raphson iterative nonlinear IK algorithm architecture, simulations and experimental implementations. Chapter 4 describes the CSA algorithm for generating smooth trajectories in Special Euclidean space. Chapter 5 provides a description of the SI-PoE trajectory generation algorithm which can satisfy multiple objectives simultaneously. This is followed by Chapter 6, where a deep RL algorithm for solving the IK problem is outlined along with simulations and experiments. Finally, Chapter 7 concludes the work with closing remarks and recommendations for future work.

2. Literature Review

This chapter presents an overview of work related to trajectory generation and planning. The objective of this review is to identify tools that can be extended and adapted to joint space trajectory generation for high degree of freedom robotic systems.

2.1. Inverse Kinematics

Trajectory generation and motion planning is an important part of robot control, which most often is carried out with the end-effector's position and orientation in mind. This is not problematic when the closed-form analytical solution is available. However, in cases where there is no such solution, the process of obtaining joint trajectories, or inverse kinematics (IK), becomes a challenging task, especially in the presence of obstacles or when minimization of effort is of importance as well. The inverse kinematics (IK) problem has been an important topic in the robotics field for a long time, and many different approaches have been demonstrated to generate joint trajectories that satisfy a specific end-effector Cartesian trajectory. As the agility of robotic manipulators becomes a crucial design consideration, which increases the number of joints, the IK problem becomes even more involved as redundancy is introduced. Thus, machine learning (ML), artificial neural networks (ANNs), or SI algorithms are attractive options to handle such highly-nonlinear problems, which is evident by the recent interest in using SI/PSO algorithms to address the IK problem.

Robotic manipulators' applications span different sectors from automotive to space, where they are used extensively in various industrial manufacturing and assembly processes, including, but not limited to, welding, sorting, handling radioactive or hazardous materials, surgery, transferring payloads in and out of space stations, etc. As Machine Learning

(ML), Artificial Neural Networks (ANNs), and Artificial Intelligence (AI) in general, matured in image recognition and vision systems, the next logical step would be to achieve full autonomy of robotic manipulators (Sridharan & Stone, 2007; Yip & Das, 2019). However, there are certain challenges that have to be overcome to achieve this goal. For instance, depending on the number of joints and links possessed by a manipulator, the mapping from Cartesian space to the robot's joint space becomes very involved, which is problematic as the tasks that a robotic arm receives are in Cartesian space, whereas the commands (velocity or torque) are in joint space. Therefore, the inverse kinematics (IK) is arguably one of the most important aspects that has to be addressed in order to achieve a fully autonomous robotic manipulator.

Generally the algorithms for solving the IK problem can be classified into three categories: analytical, iterative, and intelligent. The first category is highly limited since it can only be applied to simple robotic manipulators or specifically designed ones that have finite IK solutions. The second category can be considered as the conventional method of approaching the IK problem, where the algorithm most often aims to solve for the joint trajectory while satisfying only a single objective, such as minimizing joint effort and/or movement. Some work has proposed using Jacobian pseudo-inverse methods to prioritize tasks in the workspace (J. Park et al., 2001) and keep the joint limits within the physical bounds (Klein & Huang, 1983). The most commonly applied iterative approach is the Newton-Raphson iterative nonlinear root-finding method, which considers the IK problem as a nonlinear optimization problem (Lynch & Park, 2017).

The latter category can be described as a fresh view on the IK problem, since, in this approach, techniques that were only recently developed and applied to engineering problems are leveraged. Amongst SI techniques, the Particle Swarm Optimization (PSO) has received the most attention due to its performance on high-DoF IK problems (Collinsm & Shen, 2017). Ample literature is available on different variations of PSO applied to IK problems. One approach was to decouple the manipulator into two segments, thus approaching the IK in a bidirectional fashion (Ram et al., 2019). Another work decoupled position and orientation by applying two PSO algorithms to achieve faster convergence, and, in addition, used the inverse Jacobian to smooth the trajectories, thus achieving position control (Khan et al., 2020). Attempts to improve the performance of the artificial intelligence algorithm were made by adding a constriction factor and adaptive inertia to the PSO (Lin et al., 2016), applying a nonlinear dynamic inertia weight adjustment (Yiyang et al., 2021), and even combining PSO with Agoraphilic for obstacle avoidance (Bilbeisi et al., 2015). The scalability of the PSO to high-DoF IK problems has been explored (Collinsm & Shen, 2017). Furthermore, quantum behaved PSO was proposed as an IK solution where an improvement in performance was demonstrated (Dereli & Köker, 2020).

Even though many works have been published with different SI/PSO variants, there are certain aspects that often have not been fully addressed. These include unclear indication of how exactly inverse kinematics problem was set up and solved, unclear collision identification algorithm, absence of torque trajectories, absence of initial conditions for each SI/PSO iteration, absence of error in end-effector position/orientation, application of the algorithm on planar (2-D) manipulators, experimental evaluation almost exclusively in simulation, and etc. The objective of this work is to address the aforementioned shortcomings by demonstrating a novel SI-PoE algorithm and experimentally validating it by applying it directly to a 7 DoF Sawyer robotic manipulator. The SI-PoE algorithm is used for the cases where the end-effector trajectory and obstacles are defined a priori in the workspace. The main idea is that the IK problem is solved by the SI given multiple goals such as minimizing control effort, avoiding obstacles, and enforcing finite jerk on the end-effector. An additional Quintic Polynomial Finite Jerk (QPFJ) method of trajectory generation is also explored to demonstrate the possibility to enforce finite jerk in joint space. The PoE is used as means of identifying any collisions with the obstacles in the workspace. However, it should be noted that these goals often over-constrain the solution leading to cases where a trade-off has to be made; for example, the end-effector Cartesian trajectory accuracy might suffer in some cases if a hard finite jerk constraint is imposed on the joints.

Apart from these methods, the application of Machine Learning (ML), Artificial Intelligence (AI) and Artificial Neural Network (ANN) techniques has gained popularity in recent years. Various RL methods, such as Genetic Algorithms (GA) and Swarm Intelligence (SI) have proven to be effective in solving IK problems, even for robotic manipulators with high-DoFs. Deep Deterministic Policy Gradient (DDPG) and Normalized Advantage Function (NAF) algorithms have shown their usefulness in continuous action spaces and, more specifically, in robotic manipulation (Gu et al., 2017). Deep Q-Networks (DQN) have proven to be useful in the robotics field, where its architecture was used for vision based manipulation (Sasaki et al., 2017a; F. Zhang et al., 2015), path planning (Xin et al., 2017; Y. Yang et al., 2020), navigation (Ruan et al., 2019; W. Zhang et al., 2019), and even collision avoidance (Xue et al., 2019). However, only a few works have explored the idea of utilizing the DQN architecture to solve the IK of a high-DoF robotic systems (Guo et al., 2019; Phaniteja et al., 2017; Zhong et al., 2021), and even fewer have implemented the findings on real robotic manipulators.

This work, on the other hand, examines the use of the DQN algorithm for solving the aforementioned IK problem for the 7-DoF robotic manipulator, and implements it on a physical Sawyer robotic arm. Understandably, the DQN algorithm works better for discrete action spaces and applying it on a continuous action space as joint space control might seem counterproductive, but this work aims to demonstrate that with a modest compromise in accuracy, the DQN networks can be trained for generating complex trajectories in joint space even for a high-DoF robotic arms. The application of such methods could be designing joint space trajectories for robotic manipulator that perform routine tasks, e.g. space servicing mission or welding in a factory line, which would reduce the input required from the engineers setting up these manipulators, as they would only have to double check the trajectories generated by the network reducing their workload, saving resources and time. The simpler architecture of the DQN compared to, for instance, DDPG means that it will be easier and faster to tailor it towards specific design objectives. It is also believed that even the compromise in accuracy will be eradicated in the near future, since faster and more reliable hardware becomes more and more readily available, leading the author to believe that a simpler RL algorithm as DQN will become even more convenient and accesible. Thus, this work aims to present a RL approach based on ANNs for solving the IK problem for robotics with simulations and experimental implementation on the Sawyer 7-DoF robotic manipulator.

2.2. Analytical Methods

The IK problem can be solved analytically for simple manipulators with a few DoF and, also, for a 6-DoF arms like Programmable Universal Manipulation Arm (PUMA-type) robotic arm systems. Open-chain robotic manipulator with 6-revolute joints is a popular design, which consists of a wrist with 3-revolute orthogonal axis joints connected by an elbow joint to a shoulder with 2-revolute orthogonal axis joints. This type of robotic manipulator is demonstrated in Figure 2.1.

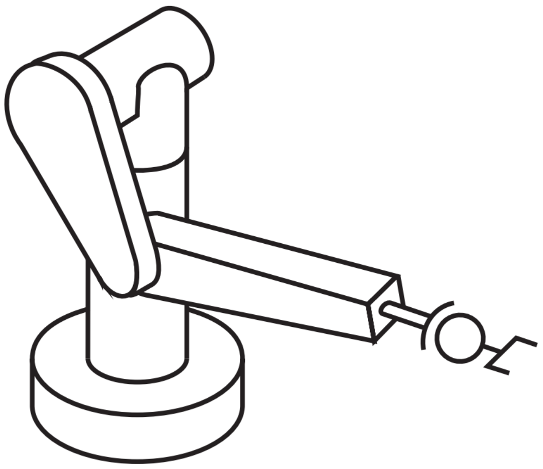

Figure 2.1 PUMA robotic arm (Lynch & Park, 2017).

Furthermore, Stanford-type arms admit analytical IK solutions. The only difference of this type of arm from the PUMA-type arms is that the revolute elbow joint is replaced by a prismatic joint.

Generally, 6-DoF open-chain manipulators have finite numbers of IK solutions. It was proven that most general 6-revolute joint robotic arns have up to 16 IK solutions (Lee & Liang, 1988; Raghavan & Roth, 1990). Some analytical methods decompose the IK problem to a set of basic screw-theoretic subproblems, called the Paden–Kahan subproblems, where the goal is to find an angle of rotation for a zero-pitch screw motion between a pair of given points (Murray et al., 2017; Paden, 1985).

As was mentioned before, analytical methods are limited to certain types of robotic manipulators, and as such for high-DoF ($n > 6$) they are rendered impractical.

2.3. Iterative Methods

The iterative approaches to the IK problem can be broadly classified into two categories: pseudo-inverse Jacobian and numerical methods. These methods became a necessity since the analytical solution to IK that was described in Section 2.2. is limited to simple manipulators with a few joints and links, and as the number of joints increases, the robotic arm will tend to have multiple postures corresponding to a single Cartesian point in space. As such redundancy is introduced, pseudo-inverse Jacobian and numerical methods, such as nonlinear root finding techniques, are normally employed.

The pseudo-inverse Jacobian based methods can be considered as traditional approaches, but they suffer from the inability to satisfy several objectives at once; for instance, minimizing torque and avoiding obstacles while tracking the desired Cartesian trajectory. Numerical methods, such as the Newton-Raphson nonlinear root finding technique are, generally, more flexible but might result in high computational cost or undesired postures, which would require hard coding the joint space constraints or adding an additional optimization algorithm.

Iterative methods generally require an initial guess for the joint space variables. The performance of the iterative method highly depends on the quality of the initial guess and, in the case where there are several possible IK solutions, e.g. high-DoF robotic manipulator, the method tends to find the solution that is “closest” to the initial guess.

Generally, each iteration would have a formulation close to the following one:

$$\dot{\theta}^{i+1} = \theta^i + J^{\dagger}(\theta^i)\mathcal{V} \tag{2.1}$$

where $J^{\dagger}$ is the pseudo-inverse of the Jacobian $J(\theta)$, and $\mathcal{V}$ is a twist that would take the current configuration to the desired one in one unit of time. An example of using the Newton-Raphson iterative method is outlined in Chapter 3.

Sometimes iterative methods diverge and fail to provide a solution due to the highly nonlinear nature of the IK problem. Many iterative methods rely on results and solution techniques from least-squares optimization (Chiaverini et al., 2016). Another drawback of such methods is that if there are multiple objectives to satisfy, e.g. obstacle avoidance in addition to end-effector trajectory tracking, the algorithm might encounter problems halfway through the trajectory as iterative methods do not have a sense of future possible postures. For instance, the algorithm might bring the posture of a robotic arm to a state where collision with an obstacle is inevitable due to the joint space constraints and past joint space solutions, which were only satisfying the tracking accuracy of the end-effector trajectory by finding the “closest” posture, not necessarily searching for a joint space trajectory that would avoid a static or dynamic obstacle in the future.

2.4. Intelligent Algorithms

This category describes more modern algorithms compared to the analytical and iterative methods. These modern tools developed during the recent advance of SI, AI, ANNs, RL, and ML are actively applied to the field of robotic control.

SI algorithms are inspired by the behavior of animals and are commonly used for solving difficult optimization problems. Generally, these algorithms are metaheuristic by nature, and prevalent ones include evolutionary algorithms (EA), particle swarm optimization (PSO), differential evolution (DE), and ant colony optimization (ACO) (Lones, 2014). SI-based techniques span different disciplines, researching potential applications ranging from controlling a large number of robots or unmanned vehicles, orbital swarm self-assembly and interferometry, planetary mapping, to even controlling nanobots for locating and eliminating cancer tumors (Lewis & Bekey, 1992). Recent research has shown that SI-based algorithms have promising performance, rivaling even conventional Jacobian-based methods when solving the IK problem (Huang et al., 2014; Rokbani & Alimi, 2013; Starke et al., 2016).

RL algorithms allow a robotic system to synthesize and improve its behavior through trial and error. This is achieved by learning a task defined by a reward function, such that when appropriate action is chosen it results in a reward, which in turn reinforces the favorable behavior. Thus, the success and effectiveness of an RL algorithm highly depend on the choice of a reward function. Another consideration that is taken into account by RL algorithms is the concept of cumulative rewards over time, which can promote positive long-term results leading to the exploration of a number of behaviors with larger sets of actions that could be missed by other methods.

2.4.1. Particle Swarm Optimization

PSO is a method that is a part of a larger family of SI algorithms, which describe social behavior of various ecosystems and animals such as bird flocks, schools of fish, etc (Eberhart & Kennedy, 1995; X.-S. Yang et al., 2013). PSO is metaheuristic by nature, quite simple to implement and provides the ability to converge to a good solution reasonably quickly (Y. Shi & Eberhart, 1999); thus, it is used in a variety of disciplines and applications. In simple terms, PSO performs an iterative search through the solution space using particles. Each particle contains parameters representing a solution, that denotes its current position in the given solution/search space, as well as velocity, which influences its position, guiding it to the most optimal solution. In general, PSO is a global optimization algorithm, and therefore can provide solutions within a large search space, but not solutions to a large degree of accuracy without significant computation.

Generally, PSO is more popular in the multi-agent swarm robotics field, where it has been used for transfer learning in multi-agent robot environments (Atyabi & Powers, 2013), multi-robot target searching in unknown environments (Dadgar et al., 2016), robotic swarms obstacle avoidance (Mohamed et al., 2010), and much more (Ab Aziz & Ibrahim, 2012; Couceiro et al., 2012; J. Yang et al., 2019). However, as PSO is an optimization algorithm at its core, it can be applied for solving nonlinear high-order problems such as IK.

2.4.2. Q-Learning and Deep Q-Learning

In contrast to PSO, Q-learning does not use a set of individual particles for locating its solution. Instead, a Q-learning agent learns through performing actions on the environment and observing results. The Q-learning algorithm by itself works by guiding a reinforcement learning agent through an environment defined by a Markov Decision Process (MDP) (Hester et al., 2017). MDP is described by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}\}$, where $\mathcal{S}$ is a set of available states, $\mathcal{A}$ is a set of actions, $\mathcal{R}$ is a set of rewards, and $\mathcal{T}$ is the transition function. Using properties known as Q-values, an agent is able to correlate actions for expected rewards, as well as to select an optimal action for a certain state. These Q-values are updated after each iteration over the problem space and calculated using the Bellman equation:

$$Q(s,a) = Q(s,a) + \alpha(\mathcal{R} + \gamma \max Q(s',a') - Q(s,a)) \tag{2.2}$$

where $Q(s,a)$ is the current Q-value, α is the learning rate, $\mathcal{R}$ is the reward for the selected action, γ is the discount factor, and $Q(s',a')$ is the estimated reward for the set of future actions. Calculating Q-values that way, storing them in a state-action table and selecting an action with the highest value guides the reinforcement learning agent to the optimal solution.

Q-Learning and its variations are commonly used for path planning (Konar et al., 2013; Low et al., 2019), multi-agent cooperation (Hwang et al., 2004; K.-H. Park et al., 2001; Sadhu & Konar, 2017), locating and grasping an object (R. Chen & Dai, 2018; James & Johns, 2016), etc. Generally, RL algorithms such as Q-Learning are attractive because they aim to solve the fundamental robotics tasks of creating an intelligent machine capable of achieving desired task through interaction with the world by acting and reacting based on sensed information and knowledge, derived from the ability to learn and improve skills autonomously.

In addition to Q-learning, the Bellman equation is also used for DQN but in a slightly modified form to include individual weights and biases of the NN (Hausknecht & Stone, 2015). Furthermore, the resultant $Q(s,a)$ is viewed as a loss function for the network. It is given as follows:

$$\mathcal{L}(s,a|\theta_i) \approx (\mathcal{R} + \gamma \max Q(s',a'|\theta_i) - Q(s,a|\theta_i))^2 \tag{2.3}$$

where θ_i represents weights and biases of the NN for a given iteration i. Note that, compared to the Bellman equation for Q-learning, the right term in Equation 6.4 is squared to denote the standard squared error loss. The overall goal of the NN is to minimize the loss. Thus, after performing the gradient descent, this loss can be calculated and resultant weights and biases adjusted accordingly.

DQN is a method that is able to provide a solution to a given problem through reinforcement learning. In contrast to PSO, DQN is more complex as it is based on two algorithms: Q-learning and Neural Networks. The AI agent takes an action for each state, performs state transition and receives a reward depending on the result. Neural Networks, on the other hand, are a set of algorithms that are used for identifying relationships between various data pieces. They are modeled after neurons and neural connections in the human brain.

The general design of NN encompasses layers of interconnected nodes, which denote neurons. Generally, several nodes serve as input layers, which are connected to one or more hidden layers. The end result is presented by the output layers. Combining the advantages of NN and Q-learning makes it quite efficient to find solutions for problems that require an iterative approach or have large solution spaces. In addition, Q-Learning in conjunction with NN allows to overcome some challenges that are common for Neural Networks, such as the large amount of data required to train the NN agent (Mnih et al., 2013). Using a reinforcement learning approach, which is iterative by nature, data can be continuously generated after each pass over the problem environment; consequently that data can be fed to the NN for it to learn and optimize the solution. One last benefit lies in NN having the ability to efficiently “remember” the data, so any new iterations and learning experiences will require less computation. Similarly to Q-Learning, DQN is used for various robotics applications including, but not limited to, path planning (Lv et al., 2019; Xin et al., 2017; Y. Yang et al., 2020), vision-based navigation and control (Sasaki et al., 2017a; F. Zhang et al., 2015), and grasping and picking (Deng et al., 2019; Joshi et al., 2020; Liang et al., 2019).

2.4.3. Deep Deterministic Policy Gradient and Normalized Advantage Functions

Processes in the real world are predominantly continuous: positions, velocities, forces and torques seldom possess a discrete range of possible values. Robotic arm manipulation is no exception as it involves complex nonlinear continuous mappings for accurate trajectory tracking.

A universal approach in AI research is to discretize a state of the world, which introduces a number of issues. In the framework of Q-Learning or DQN, for instance, the conventional approach is to store action-values in a look-up table by discretizing the continuous variables present in the solution space. If the states are discretized too coarsely, it leads to a perceptual aliasing, resulting in a loss of information and an inability to reach an optimal desired state (Gaskett, 2002). The first obvious solution to the problem of perceptual aliasing is to have a finer discretization of the continuous action space. However, it leads to a high computational cost as the number of actions increases exponentially with each extra DoF, which is also known as the curse of dimensionality (Bellman, 1966; Lillicrap et al., 2015). This makes the exploration of such vast action spaces a computationally insurmountable task.

Thus, RL algorithms that work with continuous action spaces were developed. DDPG, NAF, and Soft Actor-Critic (SAC) methods are prominent examples of model-free RL algorithms (Haarnoja et al., 2018). The main limitation associated with most model-free reinforcement approaches is that a large number of training episodes are required to find

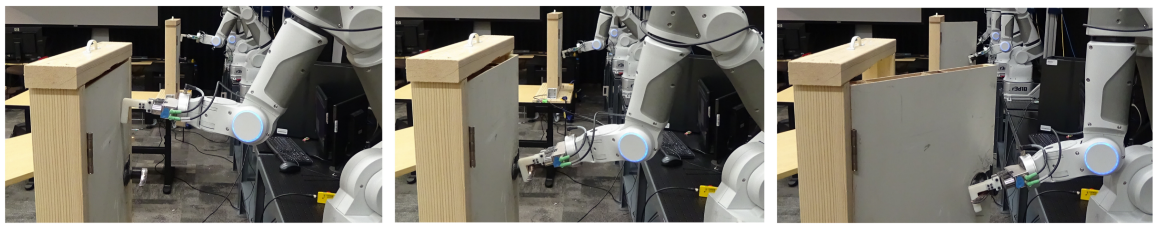

Figure 2.2 Two robots learning to open doors using a RL algorithm (Gu et al., 2017).

solutions. Depending on the problem, simpler algorithms might be more attractive due to the less complex model architecture and faster times to obtain solutions.

DDPG is a model-free, off-policy actor-critic algorithm that uses deep function approximators which can learn policies in high-dimensional, continuous action spaces. It uses four neural networks: a Q network (θ^{Q}), a deterministic policy network (θ^{μ}), a target Q network $(\theta^{Q\prime})$, and a target policy network $(\theta^{\mu\prime})$. The Q network and policy network follow the simple Advantage Actor-Critic model, but in DDPG, the Actor directly maps states to actions instead of outputting the probability distribution across a discrete action space. Both target networks slowly track the original networks, which significantly increases the stability with a slight reduction in speed (Lillicrap et al., 2015). DDPG is used for motion and path planning (Z. Li et al., 2020; Wen et al., 2018; A. Yang et al., 2021), tracking control (Liu & Huang, 2021), end-effector tasks without any policy initializations and demonstrations (Gu et al., 2017; Vecerik et al., 2017), etc. It was also demonstrated that the DDPG and NAF algorithms can benefit from having multiple robotic arms learning in parallel, which leads to a faster learning rate proportional to the number of robots (Gu et al., 2017).

NAF algorithms were developed as a continuous action spaces extension to the

Q-Learning with deep neural networks. The main idea behind NAF is to describe the Q-function in such a manner that its maximum $\operatorname{argmax}_a Q(s_t, a_t)$ is calculated analytically during the Q-Learning update (Gu et al., 2016). This is achieved by estimating a separate value function $V(s|\theta)$ and advantage term that is parameterized as a quadratic function of nonlinear features of the state:

$$Q(s, a|\theta^Q) = A(s, a|\theta^A) + V(s|\theta^V) \tag{2.4}$$

$$A(s, a|\theta^A) = -\frac{1}{2}(a - \mu(s|\theta^\mu))^T P(s|\theta^P)(a - \mu(s|\theta^\mu)) \tag{2.5}$$

where $P(s|\theta^P)$ is a state-dependent positive definite square matrix that is parametrized by a lower triangular matrix $L(s|\theta^P)$ whose entries come from a linear output layer of a NN with exponentiated diagonal terms such that $P(s|\theta^P) = L(s|\theta^P)L(s|\theta^P)^T$ (Gu et al., 2016). The rest of the NAF algorithm shares the same architecture with DQN. The NAF algorithm is considerably simpler than DDPG, and most of the time outperforms DDPG in manipulation tasks where high precision is required, making NAF a more suitable candidate for learning real world robotic tasks (Gu et al., 2016). It was used for real-time obstacle avoidance (Sangiovanni et al., 2018), manipulation tasks (Franceschetti et al., 2020; Gu et al., 2017), path planning (Sangiovanni et al., 2020), etc.

3. Newton-Raphson Iterative Trajectory Generation

This chapter presents the application of the Newton-Raphson iterative nonlinear root finding technique. It is used to solve the IK problem and generate straight-line and curviliniear end-effector Cartesian trajectories with finite jerk.

3.1. Open-loop Simulation

An open-loop simulation was performed by solving the forward dynamics given by Equation 1.2. The forward dynamics problem requires the knowledge of initial states $(\theta, \dot{\theta})$, the inverse of the inertia matrix $M^{-1}(\theta) \in \mathbb{R}^{7\times7}$, and the torque trajectory τ. For the open-loop case, the torque trajectory τ is zero for the whole duration of robot motion. The inertia matrix $M(\theta)$ is constructed column-by-column by solving the inverse dynamics problem shown in Equation 1.1 seven times, since the robot has seven joints, with angular rate $\dot{\theta} \in \mathbb{R}^7 = 0$, gravity vector $\mathfrak{g} \in \mathbb{R}^3 = 0$, external wrench $\mathcal{F}_{tip} \in \mathbb{R}^6 = 0$ and a column vector $\ddot{\theta} \in \mathbb{R}^7$ where all elements are zero except for one row $n \in [1, 2, 3...7]$, which is unity so that the n-th column vector of the inertia matrix $M(\theta)$ can be obtained. Then Equation 1.1 becomes Equation 3.1 for the first column, and is repeated for the rest of the columns by switching unity to the corresponding row:

$$\tau = M(\theta)\begin{bmatrix}1\\0\\0\\0\\0\\0\\0\end{bmatrix} \tag{3.1}$$

The $h(\theta, \dot{\theta}) \in \mathbb{R}^7$ term that combines centripetal, Coriolis, gravity, and friction forces is obtained by solving the inverse dynamics problem with $\ddot{\theta} = 0$ and $\mathcal{F}_{tip} = 0$, which simplifies Equation 1.1 to the following:

$$\tau = h(\theta, \dot{\theta}) \tag{3.2}$$

3.1.1. Inverse Dynamics Problem

The inverse dynamics problem shown in Equation 1.1 is solved using the Newton-Euler inverse dynamics algorithm described in (Featherstone, 1983, 2014; Lynch & Park, 2017), which requires joint angles $\theta \in \mathbb{R}^7$, joint angular velocities $\dot{\theta} \in \mathbb{R}^7$, joint angular accelerations $\ddot{\theta} \in \mathbb{R}^7$, gravity vector $\mathfrak{g} \in \mathbb{R}^3 = [0 \;\; 0 \;\; -9.81]^{\mathrm{T}} \mathrm{m/s^2}$, the forces and moments acting on the end-effector $\mathcal{F}_{tip} \in \mathbb{R}^6 = 0$, the relative positions of link frames in the home configuration $M_{i-1,i} \in SE(3), i \in [1, 2, 3...7]$, the spatial inertia matrices of each link $\mathfrak{G}_i \in \mathbb{R}^{6\times6}, i \in [1, 2, 3...7]$, and, finally, screw axes $\mathcal{S}_i \in \mathbb{R}^6, i \in [1, 2, 3...7]$. Since the link frames in the inertial frame given in the home configuration $M_{0,i} \in SE(3)$ are located at the center of gravity of each link, the spatial inertia matrix $\mathfrak{G}_i$ assumes the following form:

$$\mathfrak{G}_i = \begin{bmatrix} \mathfrak{I}_i & 0 \\ 0 & \mathfrak{m}_i I \end{bmatrix} \tag{3.3}$$

where $\mathfrak{I}_i \in \mathbb{R}^{3\times3}$ is the rotational inertia matrix of link i, $\mathfrak{m}_i$ is its mass, and $I \in \mathbb{R}^{3\times3}$ is the identity matrix.

With known $M(\theta)$ and $h(\theta, \dot{\theta})$ the open-loop simulation can be performed by solving the forward dynamics problem in the following form:

$$\ddot{\theta} = -M^{-1}(\theta) h(\theta, \dot{\theta}) \tag{3.4}$$

The acceleration $\ddot{\theta}(t)$ of each joint is numerically integrated using Runge-Kutta fourth-order method to obtain the states $(\theta(t), \dot{\theta}(t))$ of the robot (Hoffman & Frankel, 2018).

3.1.2. Discussion of Open-Loop Simulation Results

Figure 3.4 demonstrates the open-loop simulation of the robot with zero initial condition ($\theta(t) = 0, \dot{\theta}(t) = 0, \ddot{\theta}(t) = 0$), zero input torque on each joint ($\tau(t) = 0$), zero external forces and moments on the end-effector tip ($\mathcal{F}_{tip} = 0$), disengaged brakes, and viscous friction ($\tau_{fr} = -\eta\,\dot{\theta}(t)$) at each joint that is proportional to the angular velocity at each joint with a constant scalar viscous friction coefficient $\eta = 0.05$ Nms. The blue dot marks the trajectory of the end-effector. For the open-loop simulation, it was also assumed that the robot joints do not have joint angle limits and the robot's base is fixed at a point in space such that there are no obstacles (not even ground). The simulation time t_{sim} was 10 seconds with a timestep of $\Delta t = 0.01$ seconds.

It can be observed from Figure 3.4 that, in home configuration, the end-effector has an elevation which is then transformed to kinetic energy that accelerates the end-effector downwards following the pitching joints' motion. The energy is then slowly dissipated through the viscous friction and the end-effector moves slowly around the configuration shown when $t_{\text{sim}} = 6$ seconds.

Joint angular positions, velocities and accelerations are given in Figures 3.1, 3.2, and 3.3 respectively. It can be observed that the robotic arm tries to reach the "rest" state because of the effects of gravity and the energy dissipation due to the viscous friction in the joints. Regardless of the initial condition, the robotic arm with its brakes disengaged and zero input torque will eventually point downwards due to the presence of gravity, which is demonstrated in the open-loop response. Specifically, the pitching Joint 2 tries to reach a positive 90 deg deflection, thus pointing downwards. The angular velocities and accelerations reach large magnitudes within the initial couple of seconds, as demonstrated in Figures 3.2 and 3.3. It is worth mentioning that the robot did not reach the before-mentioned “rest” state within the first 10 seconds of the simulation; however, one can increase the simulation time or increase the damping ratio to reach that state within 10 seconds.

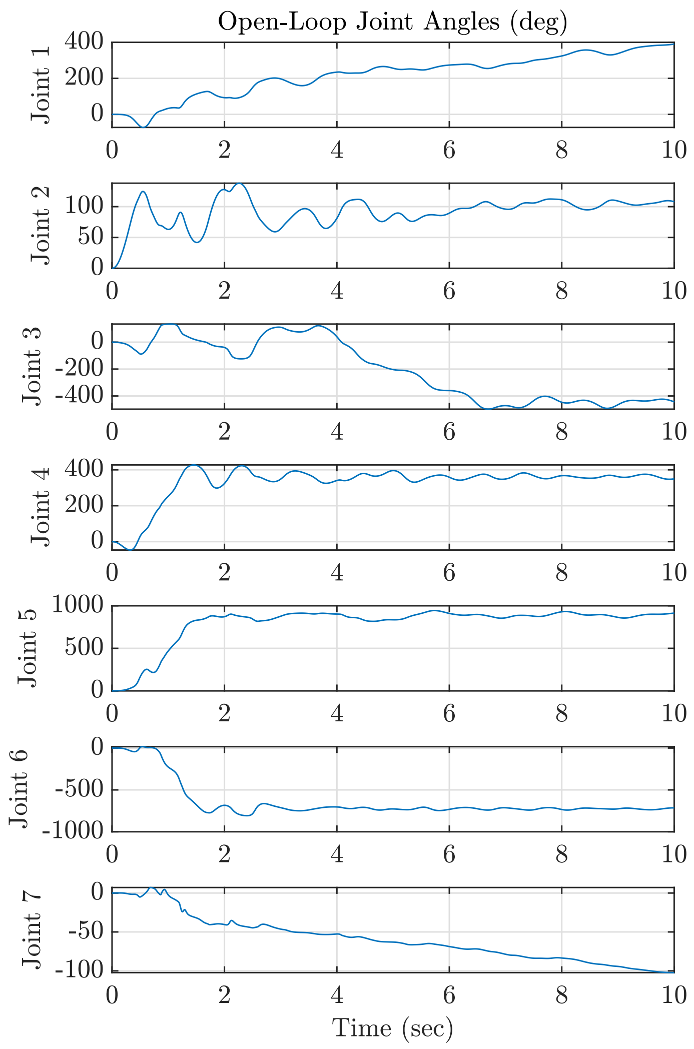

Figure 3.1 Open-loop joint angles

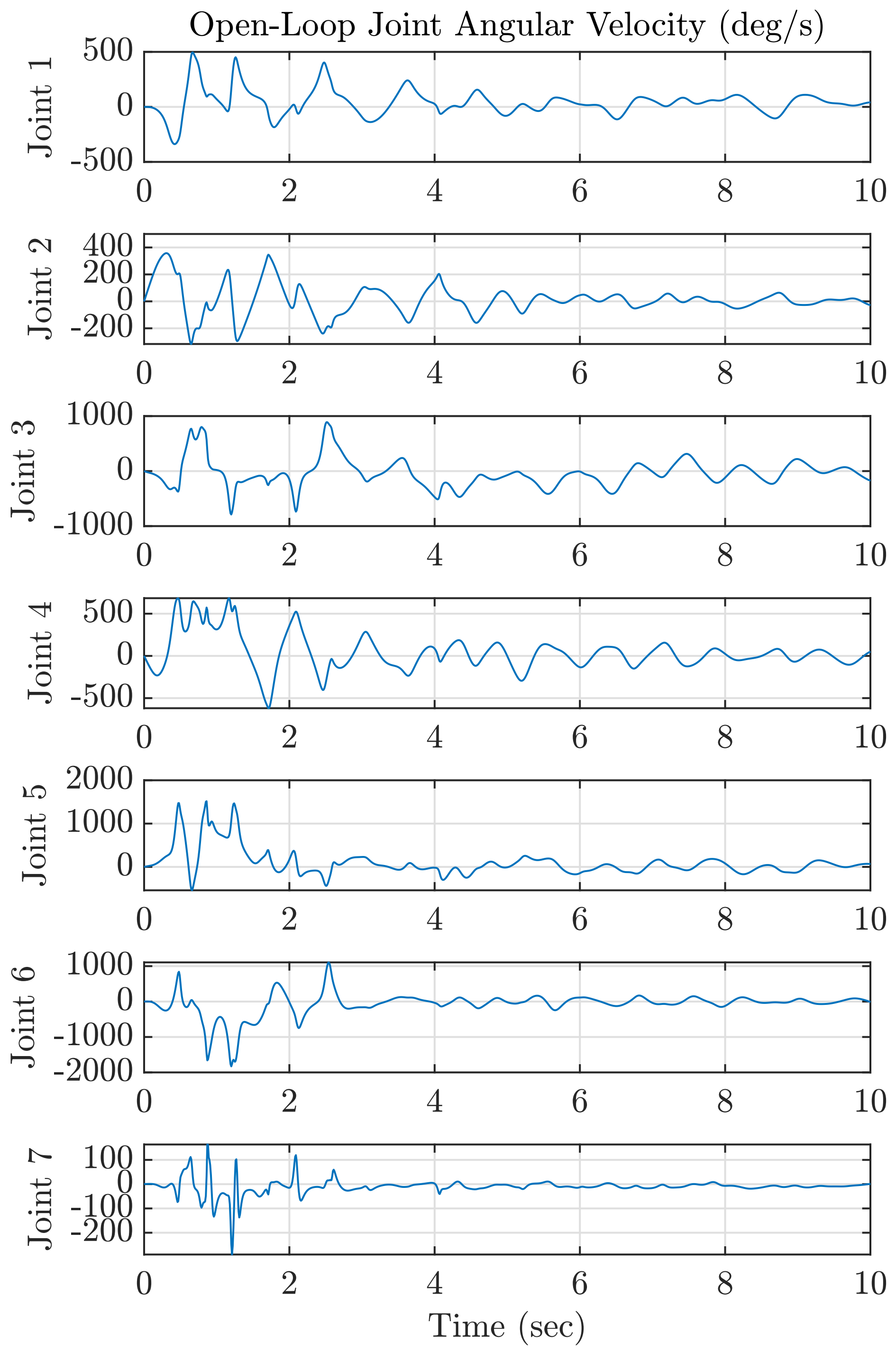

Figure 3.2 Open-loop joint angular velocities.

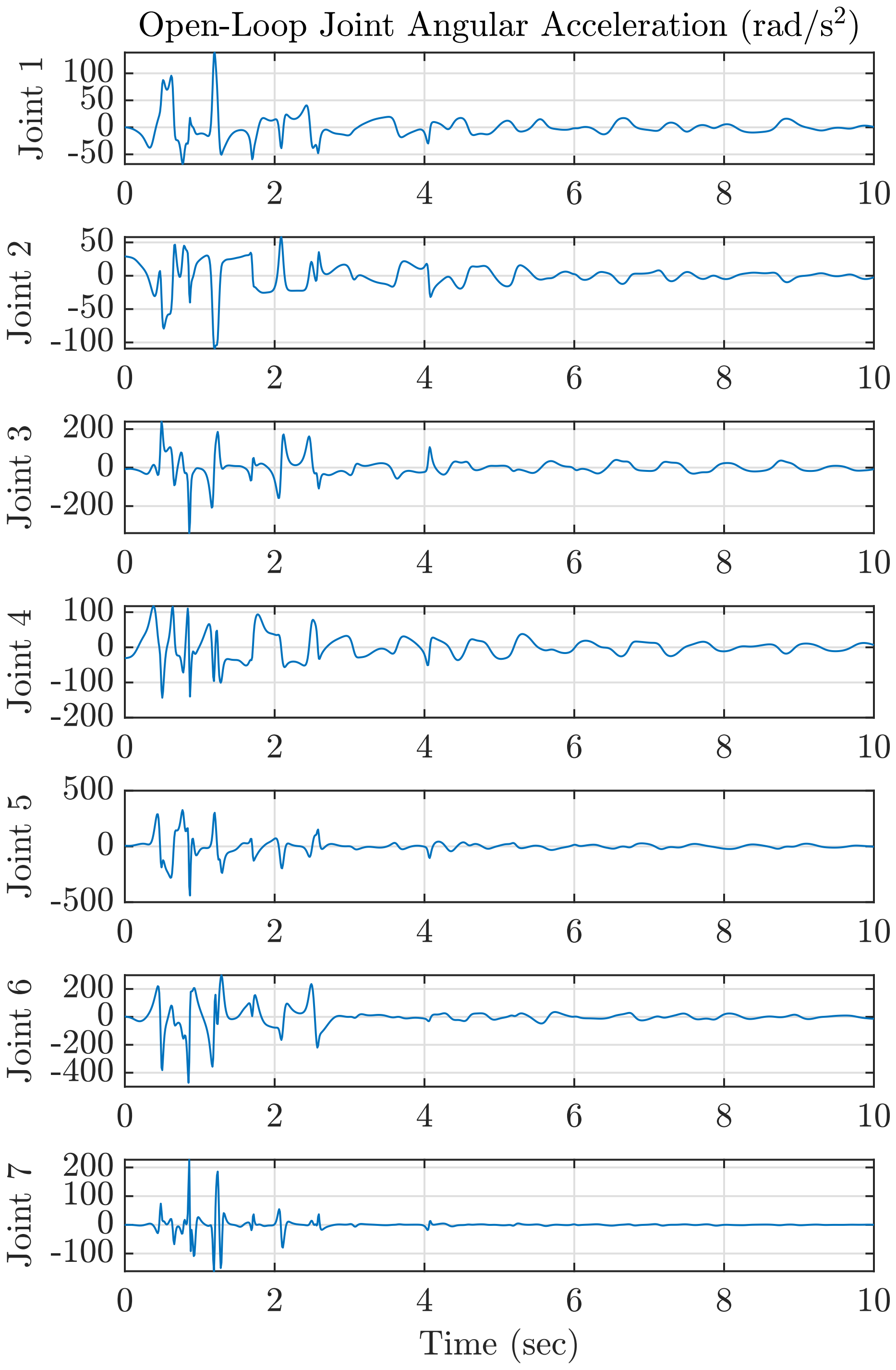

Figure 3.3 Open-loop joint angular accelerations.

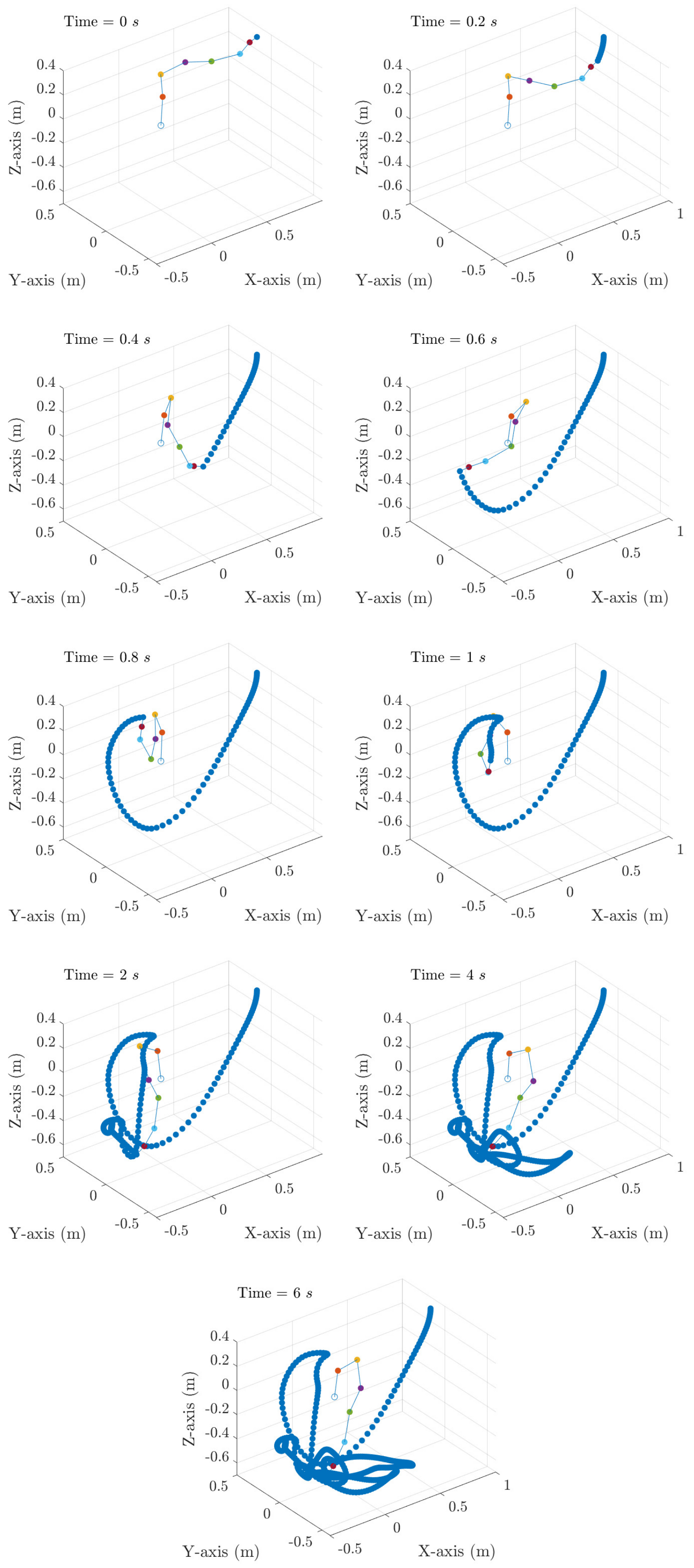

Figure 3.4 Open-loop simulation with viscous friction.

3.2. Closed-Loop Simulation (Straight Line and Curvilinear Trajectory Tracking)

In this section, an algorithm is developed to simulate a controller using the Lie algebra formulation of the dynamics of a 7-revolute (7R) joint robot. Moreover, the trajectory generation and the finite jerk constraint is explained in this section. The configuration space (C-space) of the 3R open chain manipulator can be written as: $S^1 \times S^1 \times S^1 \times S^1 \times S^1 \times S^1 \times S^1 = \mathcal{T}^7$, where $\mathcal{T}^7$ is a 7-dimensional surface of a torus in 8-dimensional space.

3.2.1. Straight Line Trajectory Description and Time Scaling

The desired end-effector trajectory $(\theta_d, \dot{\theta}_d, \ddot{\theta}_d)$ is designed in the workspace of the manipulator such that it is reachable by the manipulator's end-effector. A point-to-point trajectory is generated such that the motion of the robot starts from rest at the home configuration $(\theta_1(t_0) = 0, \theta_2(t_0) = 0, \cdots, \theta_n(t_0) = 0)$ and comes to rest at another configuration $T(t_f)$. The end-effector's exponential coordinates take the following form:

$$T_{s7}(t_0) = e^{[\mathcal{S}_1]\theta_1(t_0)} e^{[\mathcal{S}_2]\theta_2(t_0)} \cdots e^{[\mathcal{S}_7]\theta_3(t_0)} M_{s7} = M_{s7} \tag{3.5}$$

$$T_{s7}(t_f) = e^{[\mathcal{S}_1]\theta_1(t_f)} e^{[\mathcal{S}_2]\theta_2(t_f)} \cdots e^{[\mathcal{S}_7]\theta_3(t_f)} M_{s7} \tag{3.6}$$

Now, a time scaling polynomial function $s(t)$ is introduced such that it maps $t \in [0, t_f]$, $s : [0, t_f] \to [0, 1]$. More on time-scaling can be found in (Bobrow et al., 1985) and (Shin & McKay, 1985). Thus, the straight line trajectory can be defined in joint space as:

$$\theta(s) = \theta_{\text{start}} + s(\theta_{\text{end}} - \theta_{\text{start}}) \ , \ s \in [0, 1] \tag{3.7}$$

It is important to note that the straight line in joint space does not necessarily correspond to a straight line in Cartesian space. Thus, a straight line in task-space can be formulated as follows (Lynch & Park, 2017):

$$T(s) = T_{\text{start}} + s(T_{\text{end}} - T_{\text{start}})\ ,\ s \in [0, 1] \tag{3.8}$$

However, the configuration shown in Equation 3.8 does not necessarily lie in $SE(3)$. Thus the trajectory is modified as follows (Lynch & Park, 2017):

$$T(s) = T_{\text{start}} \exp(\log(T_{\text{start}}^{-1} T_{\text{end}})s)\ ,\ s \in [0, 1] \tag{3.9}$$

Still, the trajectory represented in Equation 3.9 does not produce straight-line motion in Cartesian space, rather a straight line motion with a constant screw axis that often produces a curved path in Cartesian space. In order to produce a straight-line motion in Cartesian space, rotational and translational components of the motion have to be decoupled and the motion has to be resolved in the following fashion (Lynch & Park, 2017):

$$\begin{aligned} p(s) &= p_{\text{start}} + s(p_{\text{end}} - p_{\text{start}}) \\ R(s) &= R_{\text{start}} \exp(\log(R_{\text{start}}^{T} R_{\text{end}})s) \end{aligned} \tag{3.10}$$

where, $p \in \mathbb{R}^3$ and $R \in SO(3)$ can be "extracted" from $T \in SE(3)$.

Given a desired trajectory $\theta(t)$, $t_0 \ \leq\ t\ \leq\ t_f$ the angular velocity and angular accelerations can be obtained using the chain rule:

$$\dot{\theta} = \frac{d\theta}{ds}\dot{s} = (\theta_{\text{end}} - \theta_{\text{start}})\dot{s} \tag{3.11}$$

$$\ddot{\theta} = \frac{d^2\theta}{ds^2}\dot{s}^2 + \frac{d\theta}{ds}\ddot{s} = (\theta_{\text{end}} - \theta_{\text{start}})\ddot{s} \tag{3.12}$$

Equations 3.11 and 3.12 demonstrate that the time scaling function s must be twice differentiable in order for a robot to have a well-defined acceleration (dynamics). Also, in order to avoid a discontinuous jump in acceleration, the path endpoints $\ddot{s}(0)$ and $\ddot{s}(t_f)$ are constrained to have zero acceleration, which forces the jerk to be finite throughout the whole path. However, it is crucial to note that Equations 3.11 and 3.12 can only be used for a straight line motion in joint space. In the case where the trajectory requires straight-line motion in Cartesian space, a numerical method involving Bézier Curves is used to approximate the desired end-effector joint angles trajectory θ_d and obtain $\dot{\theta}_d$, $\ddot{\theta}_d$. Adding the aforementioned constraints results in the following quintic polynomial form of time scaling:

$$s(t) = a_0 + a_1(t/t_f) + a_2(t/t_f)^2 + a_3(t/t_f)^3 + a_4(t/t_f)^4 + a_5(t/t_f)^5 \tag{3.13}$$

subject to the following constraints:

$$s(t_f) = 1 \ , \ s(0) = \dot{s}(0) = \ddot{s}(0) = \dot{s}(t_f) = \ddot{s}(t_f) = 0 \tag{3.14}$$

Equation 3.13 and 3.14 result in the following time scaling:

$$s(t) = 10(t/t_f)^3 - 15(t/t_f)^4 + 6(t/t_f)^5 \tag{3.15}$$

The plots of $s(t)$ and its derivatives are shown in Figure 3.5.

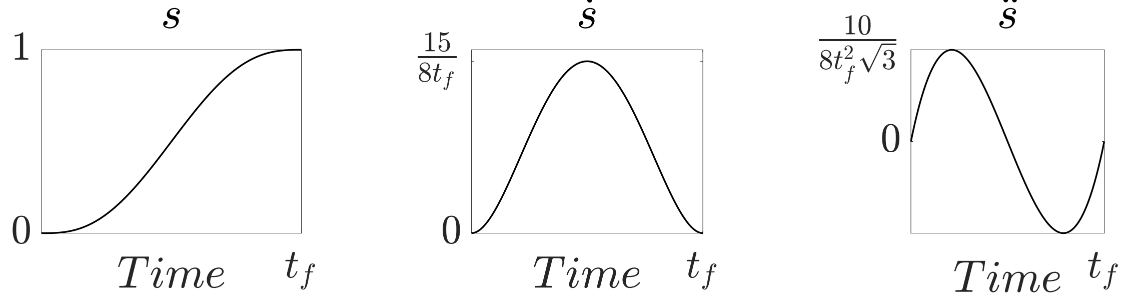

Figure 3.5 Plots of $s(t)$, $\dot{s}(t)$ and $\ddot{s}(t)$ for a fifth order polynomial time scaling.

It should be noted that the time scaling represented in Figure 3.5 does not take into account the maximum allowable angular velocity or acceleration that the actuator is capable of producing. To resolve this, the t_f parameter is set to be large enough such that the torques required at the joints after applying Equation 1.1 (inverse dynamics) do not exceed the actuator limits.

After defining the starting configuration T_{start} (which can be chosen to be the home configuration M_{s7} without loss of generality), end configuration T_{end} and the time scaling, there are 3 options for the motion. The motion can be resolved in joint space, task space or screw space. In this section, the straight line trajectory is analyzed in task (Cartesian) space.

3.2.2. Computed Torque Controller (CTC) Implementation Challenges

In the closed-loop simulation, it is assumed that each joint's motor can be actuated independently. As discussed in Section 1.1., the objective of the controller is to track a straight-line trajectory in Cartesian space with a quintic time-scaling that ensures finite jerk throughout the trajectory with zero acceleration at the endpoints. Therefore, the main idea is that the end-effector starts from the home configuration $T_{s7}(t = t_0 = 0) = M_{s7} \in SE(3)$ (Equation 3.5) and moves in a straight-line Cartesian space trajectory to some arbitrary configuration $T_{s7}(t = t_f) \in SE(3)$. Subscript $s7$ indicates that the configuration of Joint 7 is in the space (inertial) frame. For the purpose of the closed-loop simulation the initial and final configurations of the end-effector were chosen as in Equations 3.16 and 3.17.

$$T_{s7}(t_0 = 0) = \begin{bmatrix} 0 & 0 & 1 & 0.9860 \\ 0 & -1 & 0 & 0.1517 \\ 1 & 0 & 0 & 0.3170 \\ 0 & 0 & 0 & 1 \end{bmatrix} = M_{s7} \in SE(3) \tag{3.16}$$

$$T_{s7}(t_f = 10) = \begin{bmatrix} -0.8192 & 0 & 0.5736 & 0.6001 \\ 0 & -1 & 0 & 0.1517 \\ 0.5736 & 0 & 0.8192 & 1.0584 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

The generated straight-line trajectory in Cartesian space of the end-effector is illustrated

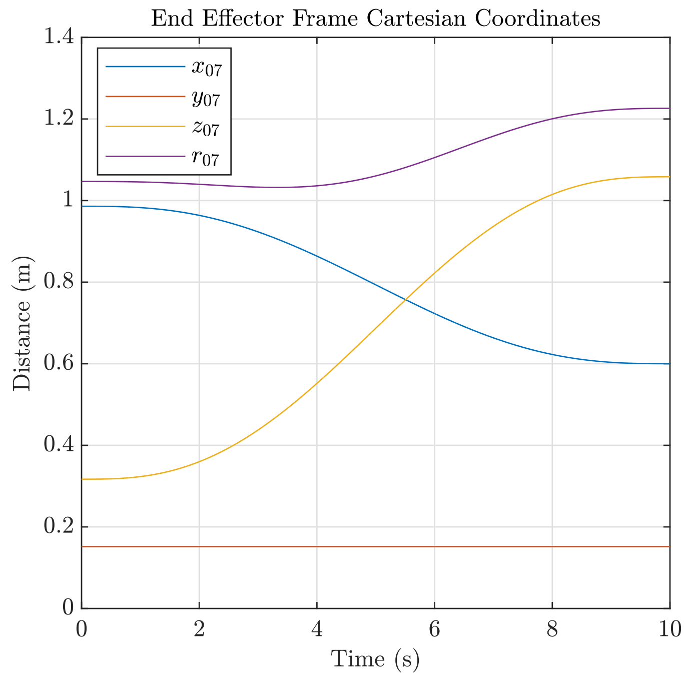

Figure 3.6 Desired end-effector displacement.

in Figure 3.6, where $x_{07}(t)$, $y_{07}(t)$, $z_{07}(t)$ are components of the end-effector configuration's $T_{s7}(t) \in SE(3)$ translational vector $p(t)$, and $r_{s7}(t)$ is its norm $(r_{s7}(t) = \|p(t)\|)$.

The challenge in this closed-loop simulation is to generate a reference desired trajectory $(\theta_d, \dot{\theta}_d, \ddot{\theta}_d)$, since the angular velocity $\dot{\theta}_d$ and angular acceleration $\ddot{\theta}_d$ are not simply the ones that can be obtained from Equations 3.11 and 3.12. The reason for this is that the straight-line trajectory of the end-effector in Cartesian space requires the solution of Equation 3.10 that involves the matrix logarithm and exponential mapping, and, in addition, the 7R robot's joint configuration can have multiple solutions for the same configuration of the end-effector (Chiaverini et al., 2016). Therefore, the first problem is to solve for the desired joint angles vector θ_d for a particular end-effector configuration using inverse kinematics (there are infinite solutions for redundant systems) (Chiaverini et al., 2016), and the second problem is to obtain the first $\dot{\theta}_d$ and second $\ddot{\theta}_d$ derivatives of the desired joint positions to generate a trajectory $(\theta_d, \dot{\theta}_d, \ddot{\theta}_d)$ for all joints that can be used for the simulation of robot dynamics.

3.2.3. Inverse Kinematics Implementation

The first of the two challenges is solved by using a numerical inverse kinematic algorithm that utilizes the Newton-Raphson iterative method of nonlinear root finding (Lynch & Park, 2017), (Chiaverini et al., 2016). This method involves an initial guess for the joint angles θ_0 and the inverse of the Jacobian matrix $J \in \mathbb{R}^{6\times 7}$ given in Equation 1.11. Since the Jacobian matrix is not square, the computation of the pseudo-inverse $J^{\dagger} = J^{\mathrm{T}}(JJ^{\mathrm{T}})^{-1}$ such that $JJ^{\dagger} = I$ is required. Given the desired end-effector configuration $T_{sd} \in SE(3)$, the iterative method can be outlined as follows:

$$\mathcal{V}_s = [\mathrm{Ad}_{T_{sb}}]\{\log(T_{sb}^{-1}(\theta^i)T_{sd})\}^{-\wedge} \tag{3.18}$$

where $\mathcal{V}_s \in \mathbb{R}^6$ is the spatial twist that takes the current calculated configuration of the end-effector $T_{sb}(\theta^i) \in SE(3)$ to the desired configuration T_{sd} if followed for a unit of time. Thus, the idea behind this iterative method is to drive $\mathcal{V}_s$ to zero, which would mean that no twist is required to move currently calculated $T_{sb}(\theta^i)$ to the desired T_{sd}. This ensures that the joint positions $\theta^i \in \mathbb{R}^7$ have converged. The update of the joint positions is given as follows:

$$\theta^{i+1} = \theta^i + J_s^{\dagger}(\theta^i)\mathcal{V}_s \tag{3.19}$$

The iterations are carried out until error tolerances $\|\omega_s\| < \epsilon_w$ and $\|v_s\| < \epsilon_v$ are satisfied or the maximum number of allowed iterations is reached. The $(-\wedge)$ symbol represents the “un-wedge” mapping that maps $se(3) \in \mathbb{R}^{4\times 4} \rightarrow \mathbb{R}^6$; the opposite is the “wedge” $(\wedge)$ mapping. Although the initial guess is known for the home-configuration, a small offset is given to the θ_0 vector so that the Newton-Raphson converges within the preset tolerances without any errors. After obtaining the $\theta(t)$ trajectory for all joints, the initial offset given to the θ_0 vector is subtracted from the entire trajectory $\theta(t)$.

3.2.4. Numerical Derivatives of Joint Angles Using Bézier Curves

Now, as the joint angles trajectory (θ_d) is obtained, the task is to obtain the first and second derivatives such that the dynamics problem is well-posed $(\theta_d, \dot{\theta}_d, \ddot{\theta}_d)$. Therefore, the derivatives are obtained numerically using Bézier Curves. The detailed algorithm is outlined in (Pastva, 1998), and it is important to mention that, in this work, Bézier Curves were only used to approximate the trajectory in the vertical direction.

The accuracy of the Bézier Curve fits was quantified using the Mean Arctangent Absolute Percentage Error (MAAPE), given in (Kim & Kim, 2016), because the trajectory of the end-effector is very close to zero in joint space, rendering Mean Absolute Percentage Error (MAPE) ineffective. The MAAPE method was implemented in the following way for a particular order of Bézier Curve:

$$\mathrm{MAAPE}_i^j = \frac{1}{N}\sum_{k=1}^{N} \arctan\left(\left|\frac{\theta_{di}(t_k) - \theta_{bi}^j(t_k)}{\theta_{di}(t_k)}\right|\right) \tag{3.20}$$

$$N = t_f/\Delta t \tag{3.21}$$

where superscript (j) indicates the order of the Bézier Curve, subscript (i) refers to the joint number, θ_{di} is the desired i-th joint angle trajectory, θ_{bi}^j is the corresponding j-th Bézier Curve fit approximation at discrete time (t_k), and N is the number of steps. Figure 3.7 demonstrates that the MAAPE reduces significantly as the Bézier Curve fit order is increased; even the fifth-order Bézier Curves well approximate the trajectories of all seven joints. The $(\dot{\theta}_d, \ddot{\theta}_d)$ were obtained by differentiating the Bézier Curves as outlined in (Pastva, 1998). The results for joints 1 through 6 are demonstrated in Figure 3.8.

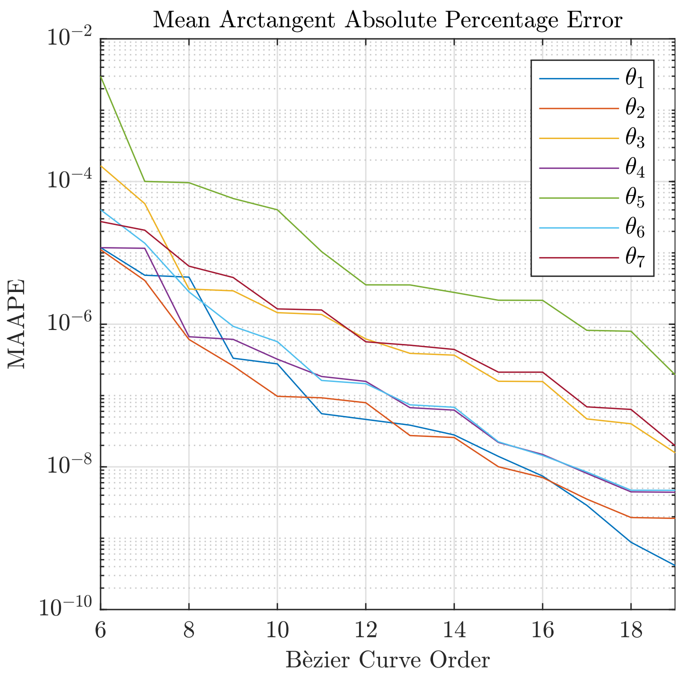

Figure 3.7 MAAPE vs. Bézier curve order.

As can be seen from the figures, the velocity is zero at the endpoints of the trajectory for all joints. However, it can be observed that the acceleration is non-zero for most of the joints at the beginning and at the end of the motion. This can be explained by the fact that the constraint on finite jerk and zero acceleration at the trajectory endpoints was only imposed on the end-effector and not necessarily on the joints themselves. Moreover, it is important to mention that joint trajectories $(\theta, \dot{\theta}, \ddot{\theta})$ were computed offline for all joints given only prior knowledge of initial (Equation 3.16) and arbitrary final (Equation 3.17) configurations of the end-effector.

3.2.5. CTC Implementation and Introduction of Uncertainties

After the desired trajectory $(\theta_d, \dot{\theta}_d, \ddot{\theta}_d)$ was obtained, the following feedforward plus feedback linearizing controller, also known as the computed torque controller, was

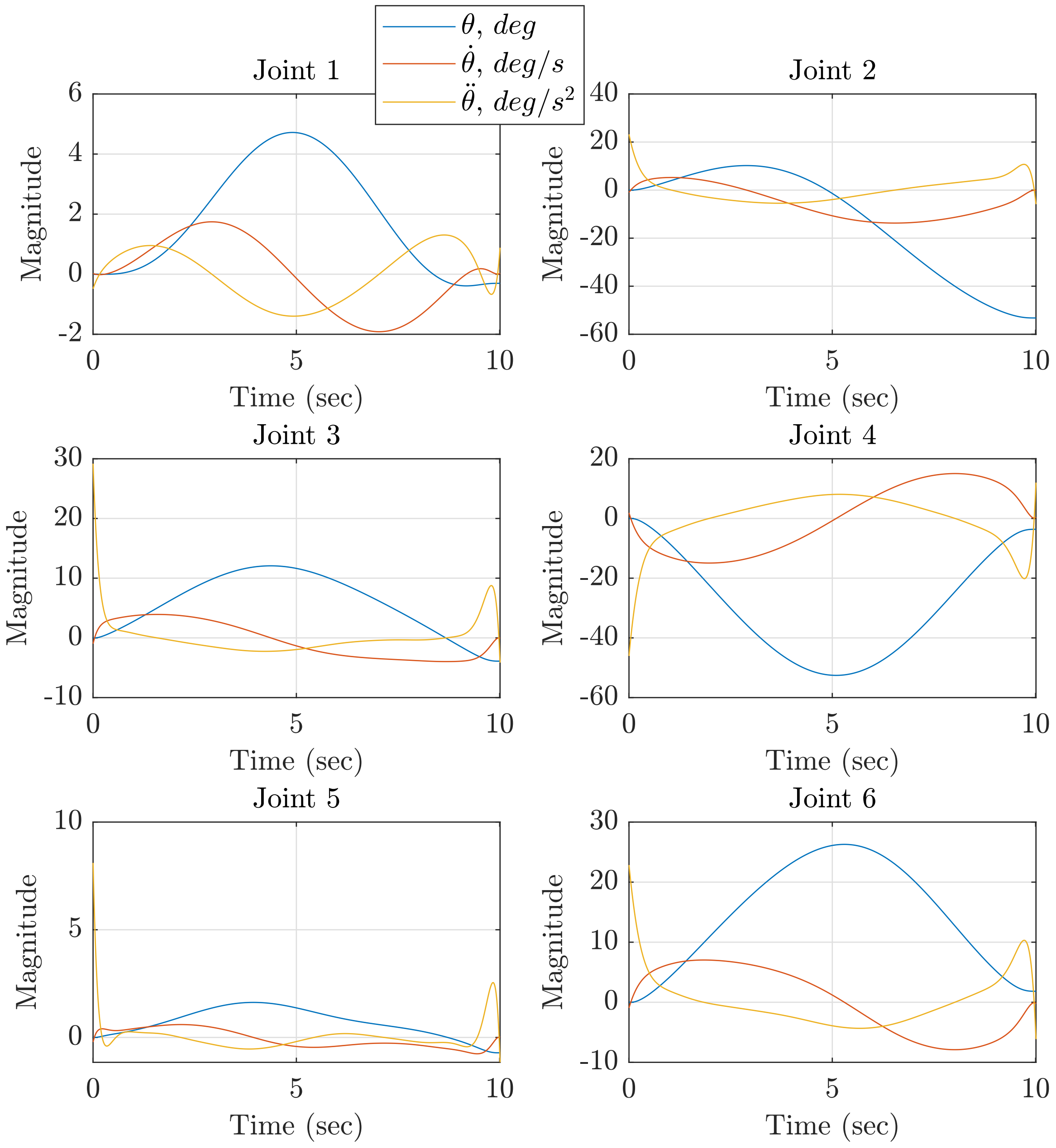

Figure 3.8 Desired trajectory $(\theta_d, \dot{\theta}_d, \ddot{\theta}_d)$ of Joints 1 through 6.

implemented (Lynch & Park, 2017; Markiewicz, 1973; Paul, 1972; Raibert, 1978):

$$\tau = \tilde{M}(\theta)\left(\ddot{\theta}_d + K_p\theta_e + K_i\int \theta_e(t)dt + K_d\dot{\theta}_e\right) + \tilde{h}(\theta, \dot{\theta}) \tag{3.22}$$

$$\theta_e(t) = \theta_d(t) - \theta(t) \tag{3.23}$$

where $K_p = k_pI$, $K_i = k_iI$ and $K_d = k_pI$ are 7×7 gain matrices with nonnegative scalar diagonal elements $k_p = 50$, $k_i = 20$, and $k_d = 10$. The tilde sign above the configuration dependent mass matrix $\tilde{M}(\theta)$ and lumped forces $\tilde{h}(\theta, \dot{\theta})$ matrix represents that there might be some uncertainties in the parameters of the system, such as the locations of the center of gravity, locations of screw axes, mass and inertia properties of the links, and gravity vector. Random uncertainties up to 2% were given to the position vectors of the relative configuration matrices $M_{i-1,i} \in SE(3), i \in [1, 2, 3...7]$ and gravity vector $\mathfrak{g} \in \mathbb{R}^3$.

A robust controller will compensate for these uncertainties, and the controller given in Equation 3.22 demonstrates a degree of robustness as shown in Figures 3.13 and 3.14 which demonstrate the ideal desired and simulated actual joint trajectories. The trajectory with uncertainties is still very close to the desired trajectory and the joint angle errors are the largest for the joints that are the closest to the end-effector, making the position (Cartesian space) error minimal.

The simulated trajectory in Cartesian space after applying the control law without uncertainties is shown in Figure 3.9, which demonstrates that the computed torque controller achieves the goal of tracking a straight-line in Cartesian space. The plot of joint angle errors is demonstrated in Figure 3.12. It should be noted that the errors in joint angles are on the order of a tenth of a degree, reaching a maximum value of $0.2°$. Corresponding errors in the end-effector position are presented in Figure 3.11 where $x_e = x_d - x$, $y_e = y_d - y$, $z_e = z_d - z$ and $r_e = r_d - r$. Position errors are on the order of 2 millimeters towards the end of the motion and reach a maximum error of $y_e = 6\text{mm}$. However, the error in the magnitude of the position vector $p(t)$ barely exceeds 2mm.

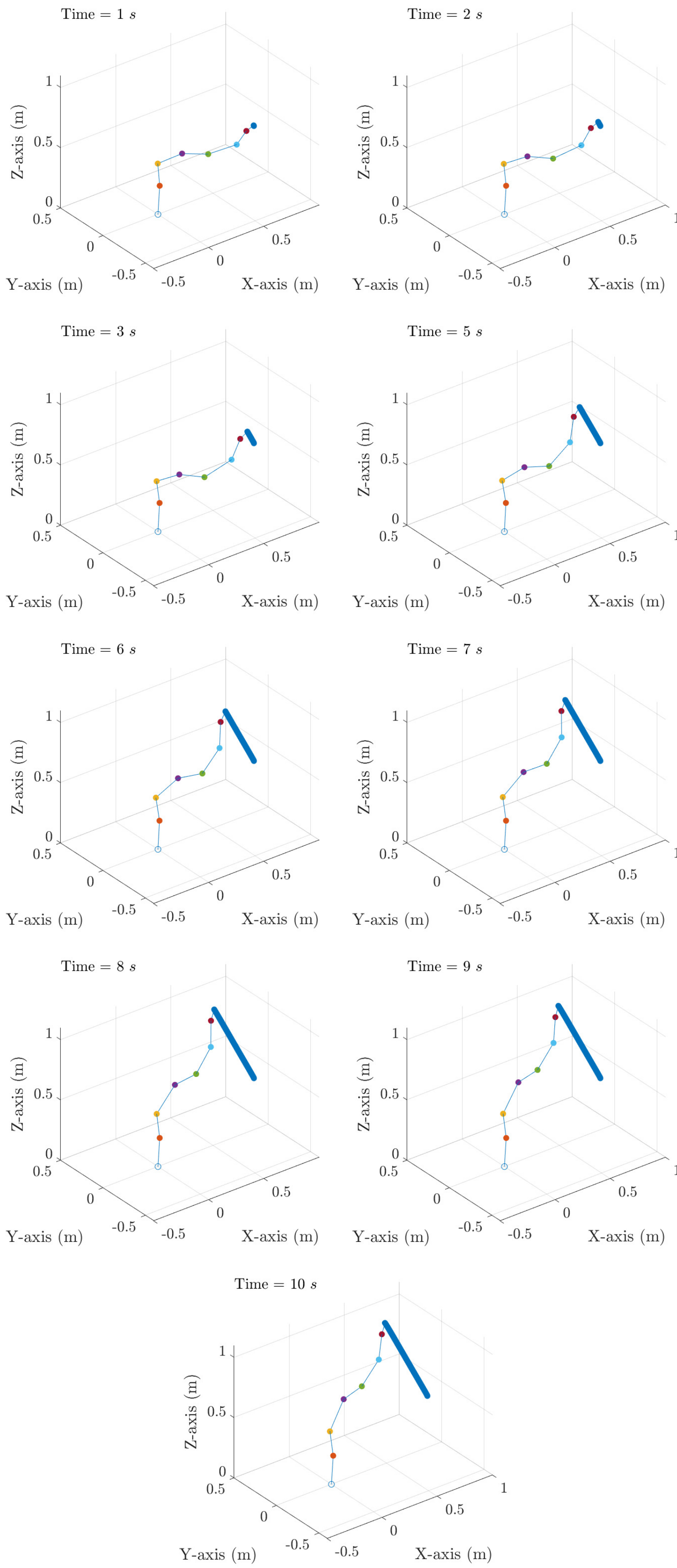

Figure 3.9 Closed-loop simulation, $t_{\text{sim}} = 10$ seconds, and $\Delta t = 0.01$ seconds.

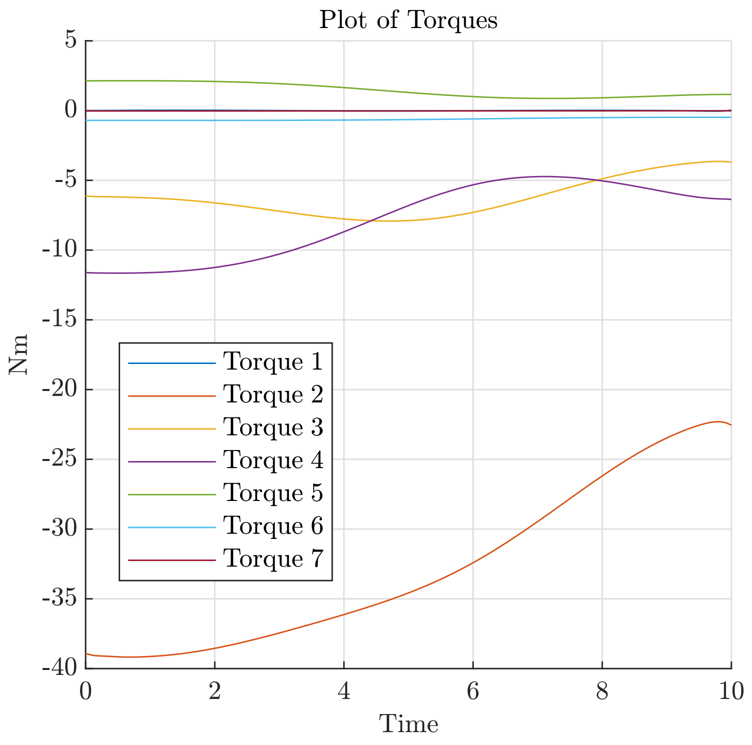

Figure 3.10 Torque trajectories, all joints.

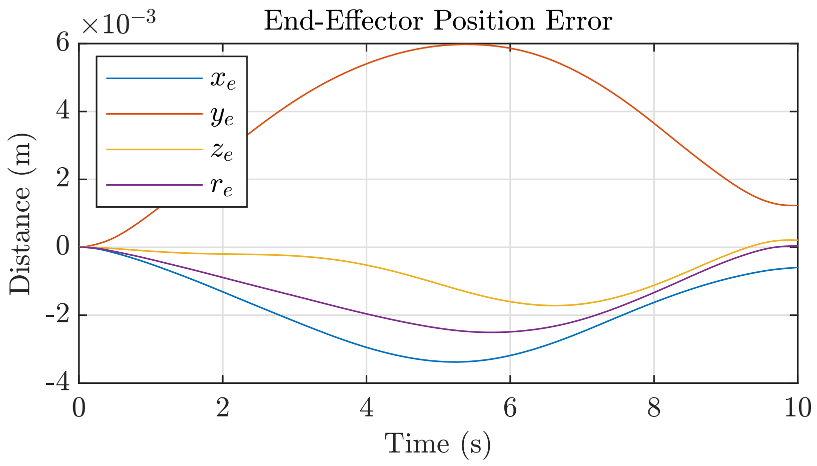

Figure 3.11 Errors in end-effector position, simulated trajectory without uncertainties.

3.2.6. Comparison of D-H and PoE: Sinusoidal Trajectories

Using this recursive method the torques for each joint can be calculated using the D-H parameters approach (Korczyk et al., 2021) or PoE approach described by Equation

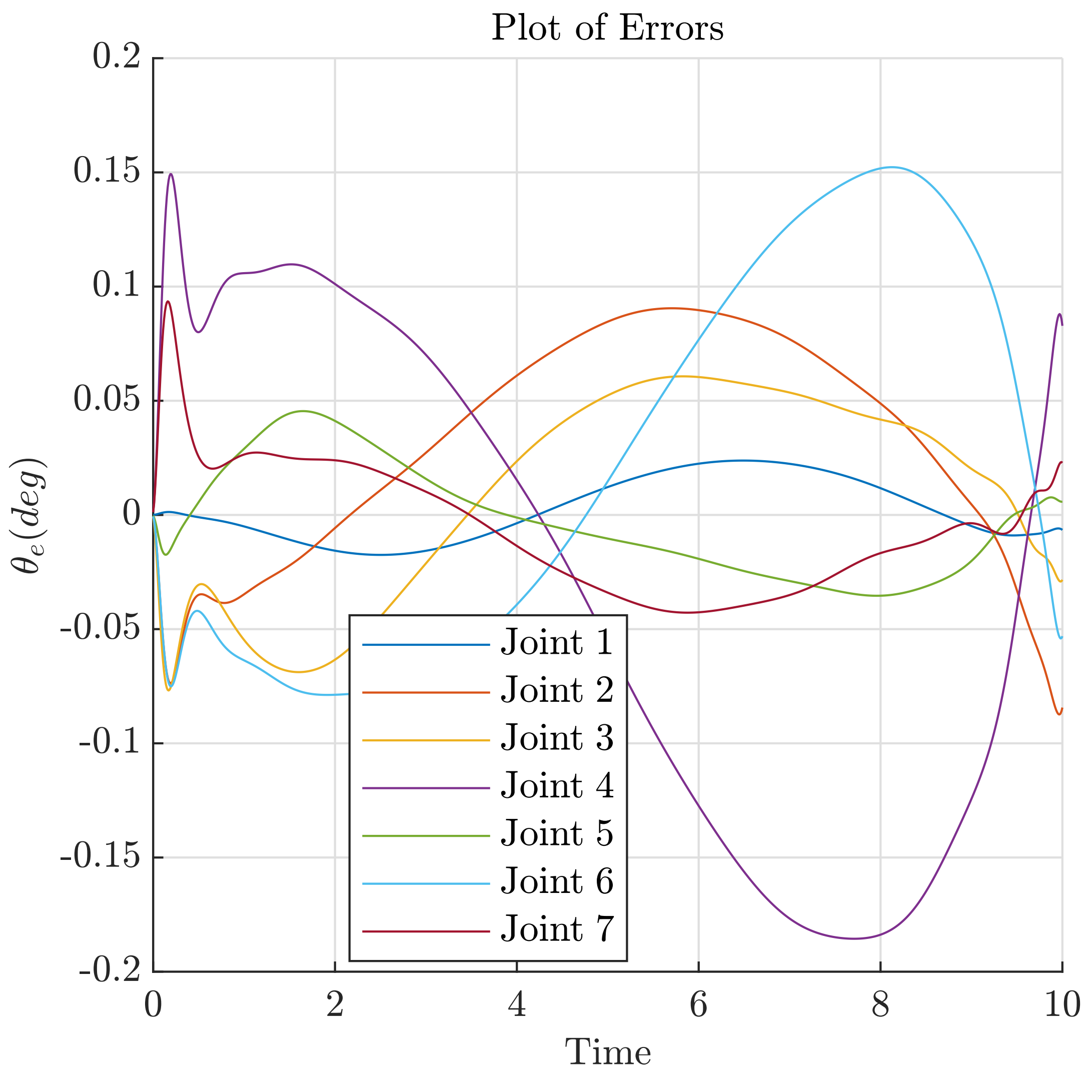

Figure 3.12 Errors in joint angles, simulated trajectory without uncertainties.

1.1. To generate the torque and the angular positions, velocities and accelerations of each joint must be specified. The specific trajectories are provided as functions of time as seen in Table 3.1, which results in the torques plotted in Figures 3.15 and 3.17. Modeled in SimScape^{TM}, the numerical simulation of the dynamics described analytically by the D-H parameters model generates identical results as shown by the subtraction of the results of the D-H parameters method from those of the numerical simulation (Figure 3.16).

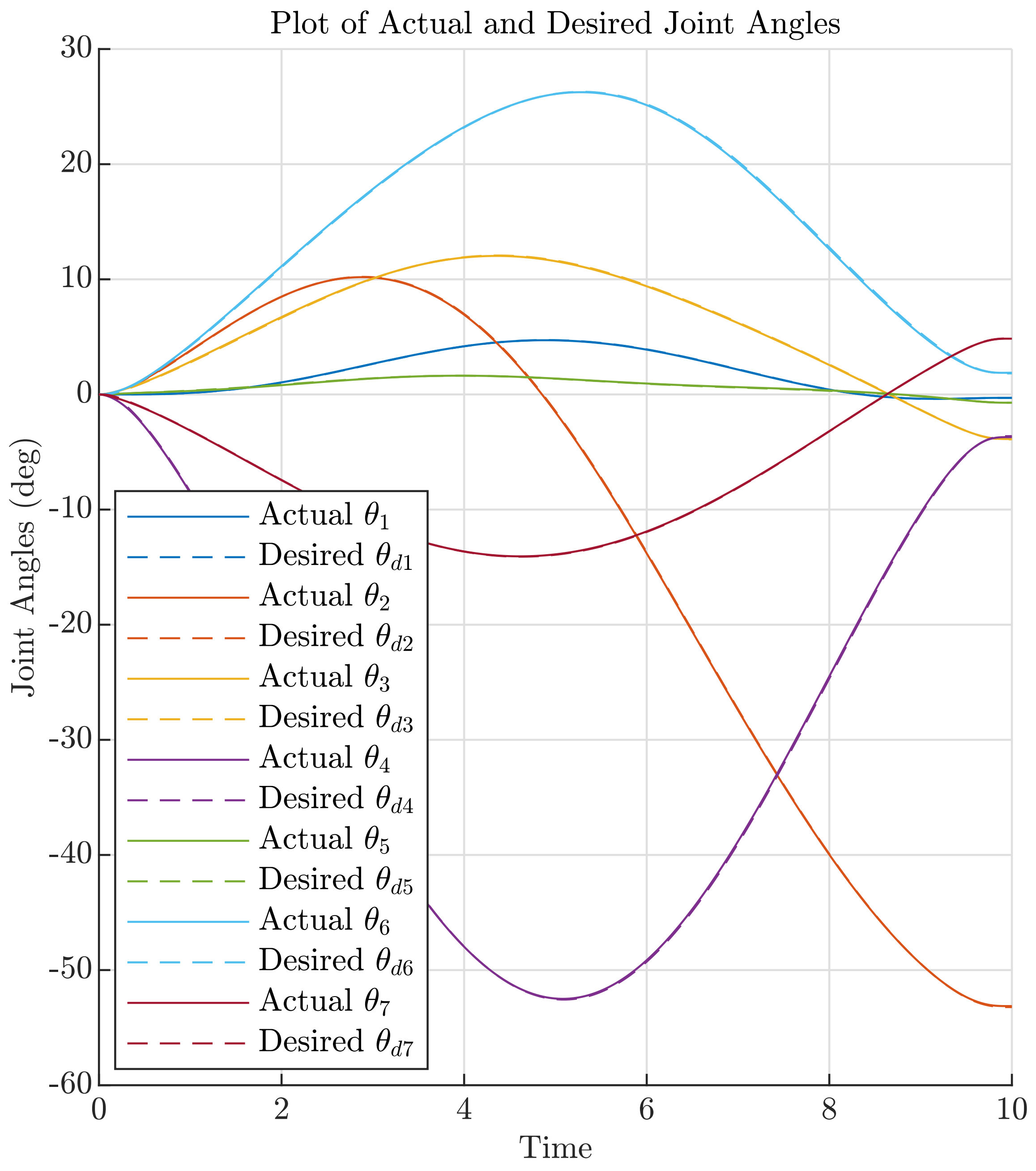

Figure 3.13 Simulated trajectory without uncertainties.

The PoE method, on the other hand, shows a different torque profile as demonstrated in Figure 3.18, which indicates that the SimScape^{TM} and D-H approaches calculate the torque profiles in a different way. This was performed to analyze the veracity of both mathematical approaches, D-H and PoE methods performed completely coded using

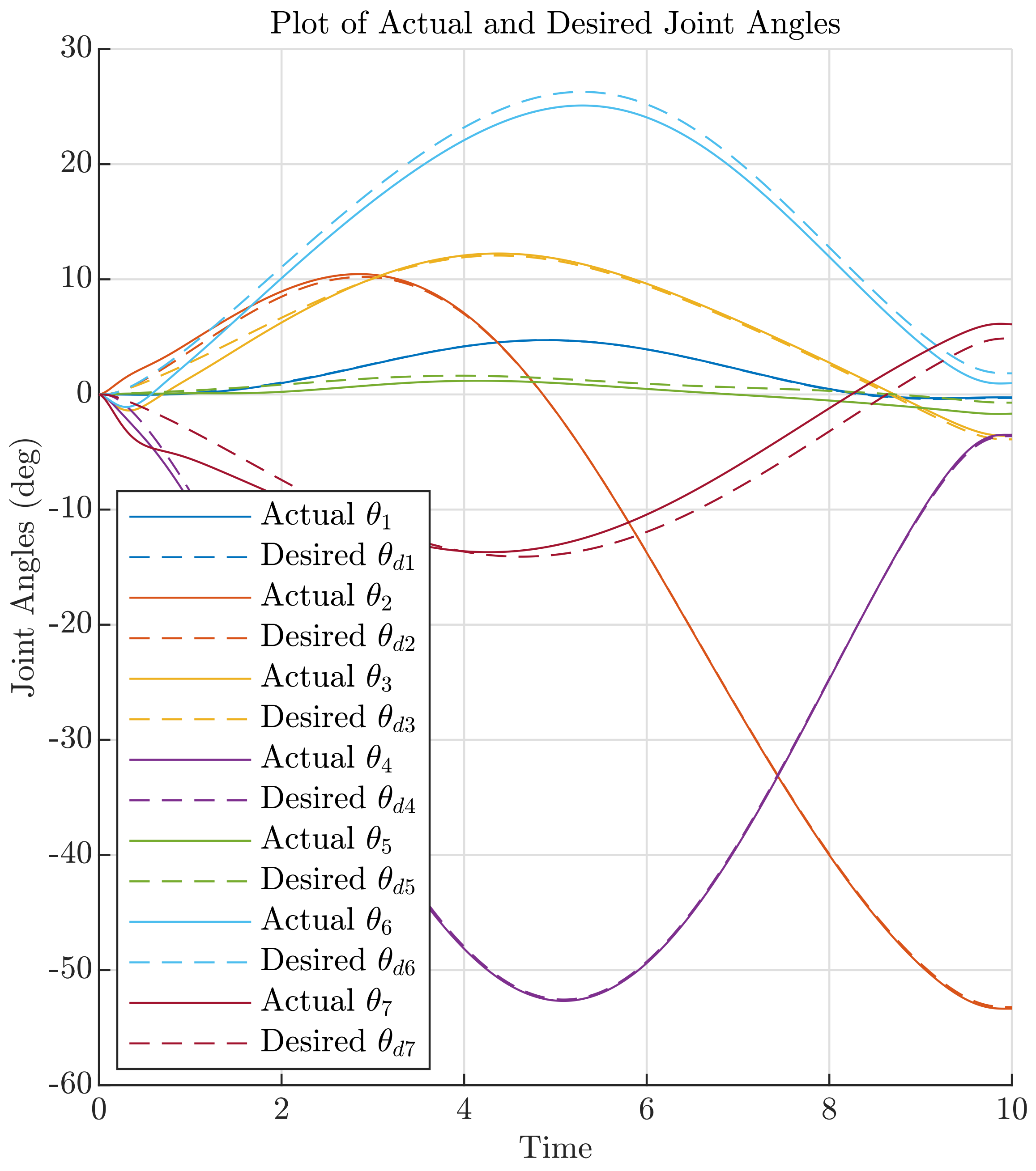

Figure 3.14 Simulated trajectory with uncertainties.

functions developed exclusively from the code editor in MATLAB. The SimScape^{TM} numerical method described the algorithm graphically using the Simulink environment through a modular block approach.

Table 3.1

Joint Trajectories

Joint	Angular position	Angular velocity	Angular acceleration
1	sin(t)	cos(t)	-sin(t)
2	cos(t)	-sin(t)	-cos(t)
3	sin(t)	cos(t)	-sin(t)
4	cos(t)	-sin(t)	-cos(t)
5	sin(t)	cos(t)	-sin(t)
6	cos(t)	-sin(t)	-cos(t)
7	sin(t)	cos(t)	-sin(t)

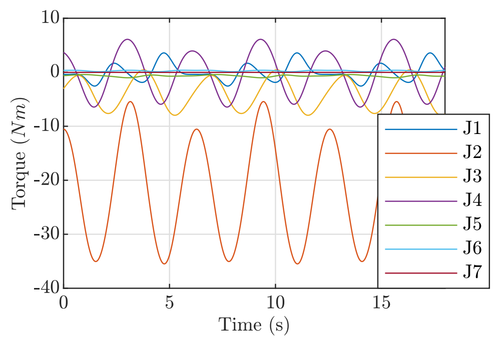

Figure 3.15 Joint torques generated by D-H parameters method.

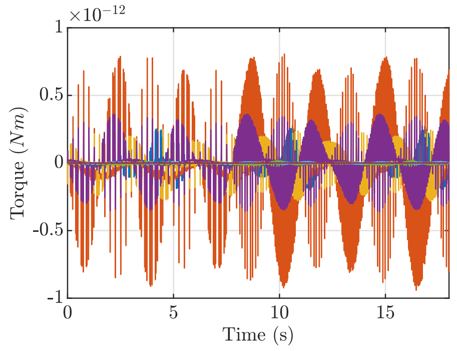

Figure 3.16 Relative error propagation between D-H and SimScape^{TM}.

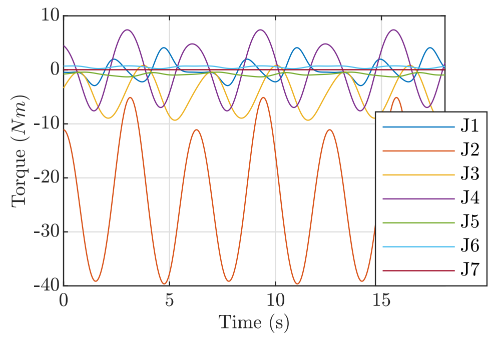

Figure 3.17 Joint torques generated by PoE method.

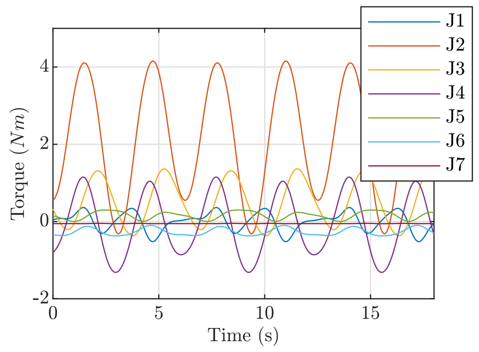

Figure 3.18 Relative error propagation between PoE and SimScape^{TM}.

The joint torque profiles generated using the recursive method (Figure 3.15) and PoE inverse dynamics method (Figure 3.17) produce different torques throughout the trajectory defined in Table 3.1. Joint 2 shows the most discrepancy where the difference between PoE and D-H torque trajectories reach a maximum value of $4\ Nm$. This could be caused by the assumptions made when modeling the links using D-H parameters when typing the code and modeling in Simulink. More information on these discrepancies will be detailed in Section 3.3.1.

3.3. Experimental Implementation

The finite jerk straight-line Cartesian trajectory was implemented on the Sawyer robotic arm using the Ubuntu-Robotic Operating System (ROS) environment. The joint space trajectories were communicated through the Sawyer Python API to the robotic arm running SDK mode. Experimental joint angles, velocities, and efforts (torques) were recorded. The recorded experimental data had noise, though the end-effector accurately tracked the desired Cartesian trajectories.

3.3.1. Curvilinear Trajectory

To ensure the validity of the D-H parameters and the PoE approaches, an experiment was conducted to compare the simulated torque trajectories with experimental ones. The comparison proved the accuracy of both methods, and it was observed that the PoE approach is more accurate than the D-H parameters approach. The trajectory for these tests was generated by commanding the robot to move from its zero (or rest) position, in which all links were configured to extend to the robot's furthest reach in the x-direction. Afterward, all links were configured to extend to the robot's furthest reach directly to the upward pose in the z-direction. The joint angles, angular velocities, angular accelerations, and torque profiles corresponding to this motion are shown in Figures 3.19-3.22, which results in a curvilinear trajectory of the end-effector in Cartesian space. The simulated torque profiles of the D-H and PoE approaches are shown in Figures 3.23 and 3.24 respectively.

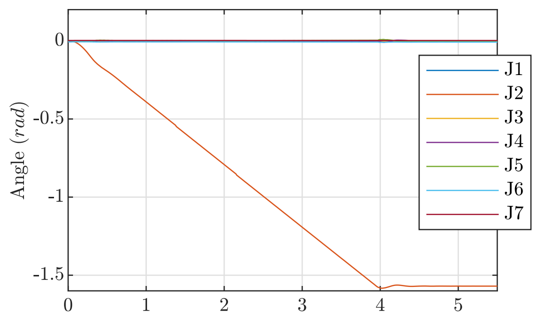

Figure 3.19 Experimental joint angles.

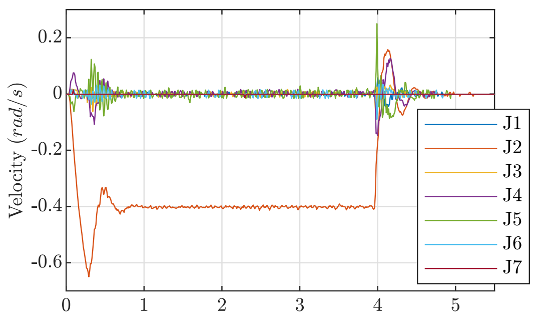

Figure 3.20 Experimental joint velocities.

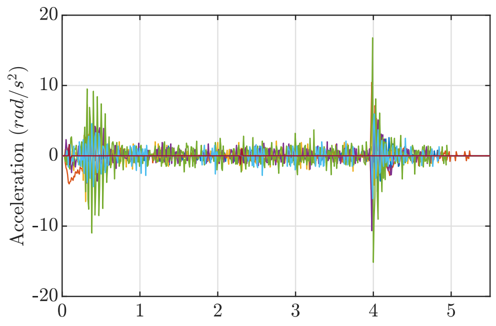

Figure 3.21 Experimental joint accelerations.

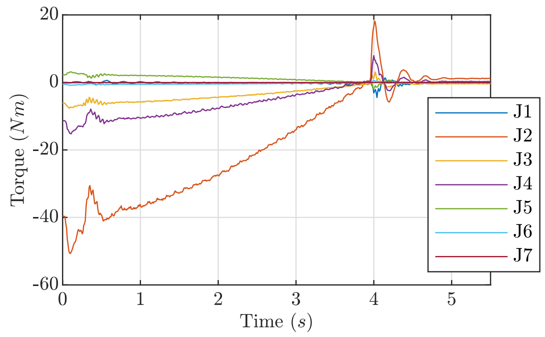

Figure 3.22 Experimental joint torques.

The error between the models and the robot can be determined by subtracting the simulation profiles from the robot generated torque profiles, as shown in Figures 3.25 - 3.31. The PoE error (shown in red) is less than the D-H parameters error on all of the joints except the first joint, where both methods demonstrated exactly the same degree of accuracy. It is worth mentioning that the PoE method exhibits exceptional accuracy in all pitching joints (joints 2,4, and 6), where, in contrast, the D-H parameters method demonstrates a high margin of error up to 7 Nm in joint 2 (Figure 3.26), which is the largest error.

Joint two is the first pitching joint, meaning it is the first joint that resists gravitational effects while moving the subsequent masses. As joint number two is required to move such a large amount of mass, it is expected that this joint will most likely result in the largest torque, which is confirmed by the prediction of the D-H and PoE algorithms (Figures 3.23 and 3.24) and shown by the Sawyer torque profile (Figure 3.22). While the profiles appear to be similar in shape, the magnitudes, however, do not match exactly.

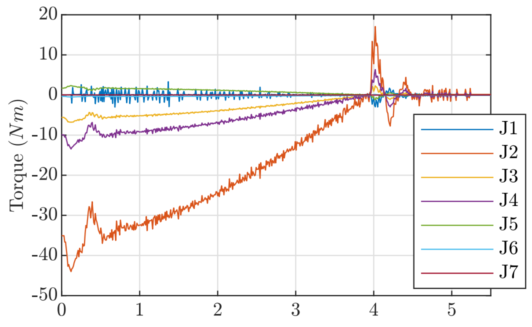

Figure 3.23 Simulated torque profile generated by D-H method.

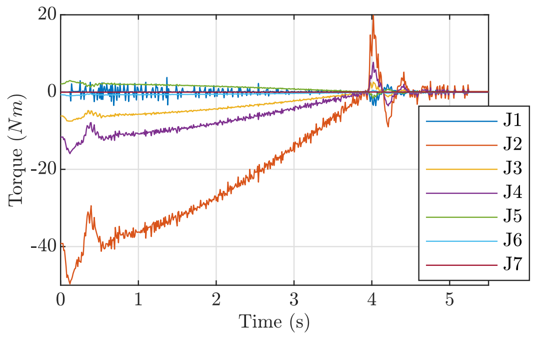

Figure 3.24 Simulated torque profile generated by PoE method.

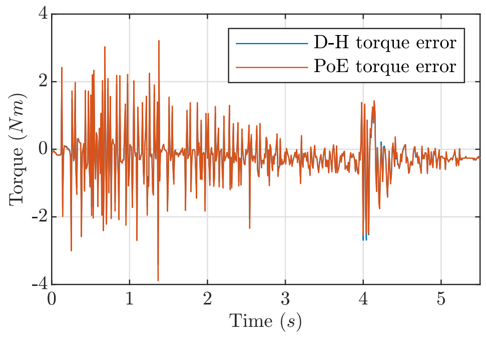

Figure 3.25 Joint 1 error.

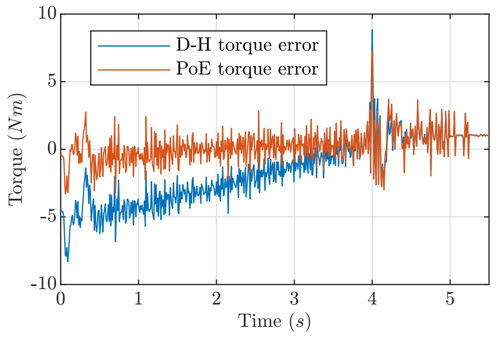

Figure 3.26 Joint 2 error.

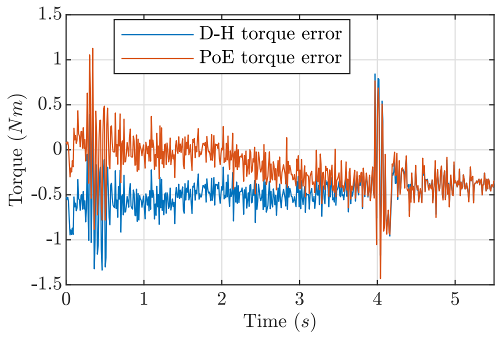

Figure 3.27 Joint 3 error.

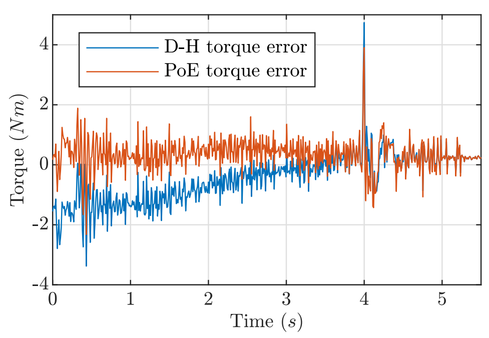

Figure 3.28 Joint 4 error.

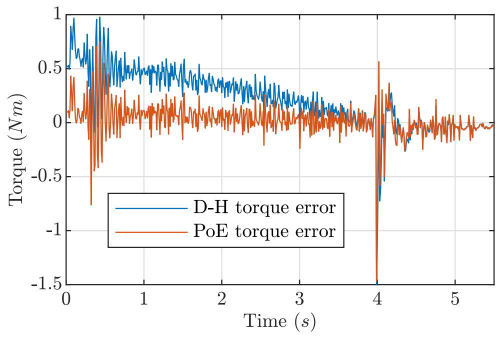

Figure 3.29 Joint 5 error.

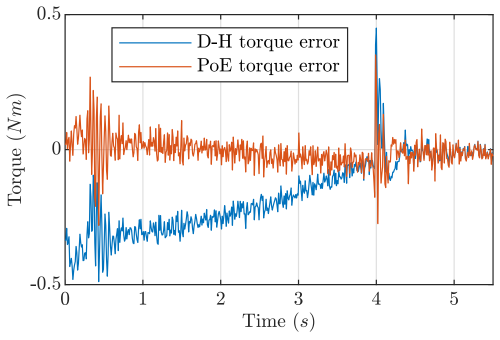

Figure 3.30 Joint 6 error.

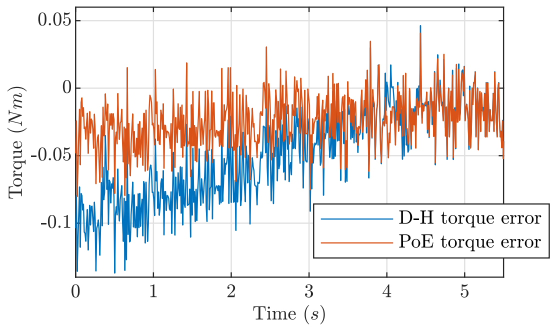

Figure 3.31 Joint 7 error.

3.3.2. Straight-line Trajectory

The Cartesian straight-line trajectory was tested on the robotic arm multiple times to confirm that it is repeatable. All of the 6 experimental runs produced almost identical joint space angles, velocities and torques with some negligible variation in the noise.

Although experimental torque profiles exhibit some noise, they follow the desired profiles well as demonstrated in Figure 3.32. Throughout the trajectory, the difference between the experimental torque profiles and the ones obtained using the inverse dynamics algorithm is around 2 Nm most of the time, with rare spikes up to 5 Nm lasting for a couple of milliseconds. The joint space trajectories almost match the desired trajectories, with the exception of joint 6 which was experiencing unusual behavior while following the simulated profile that can be observed as a jagged line in Figure 3.33. The difference between the experimental and simulated joint angles occasionally reaches 0.01 rad for all of the joints except joint 6, where the error spikes up to 0.04 rad several times, which is abnormal as demonstrated in Figure 3.34. Despite the aberrant behaviour of joint 6, the Cartesian error of the end-effector does not exceed about 6.4 mm in any of the three axes as shown in Figure 3.35.

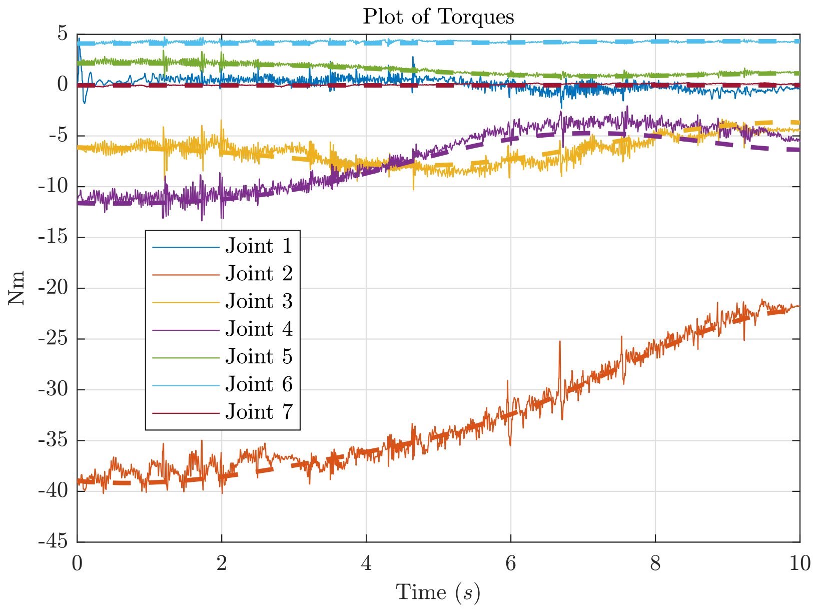

Figure 3.32 Experimental (solid) and simulated (dashed) torque trajectories.

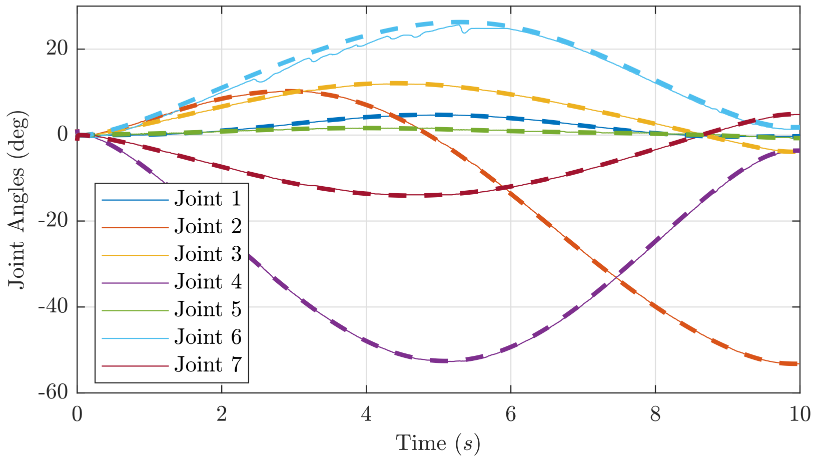

Figure 3.33 Experimental (solid) and simulated (dashed) joint angle trajectories.

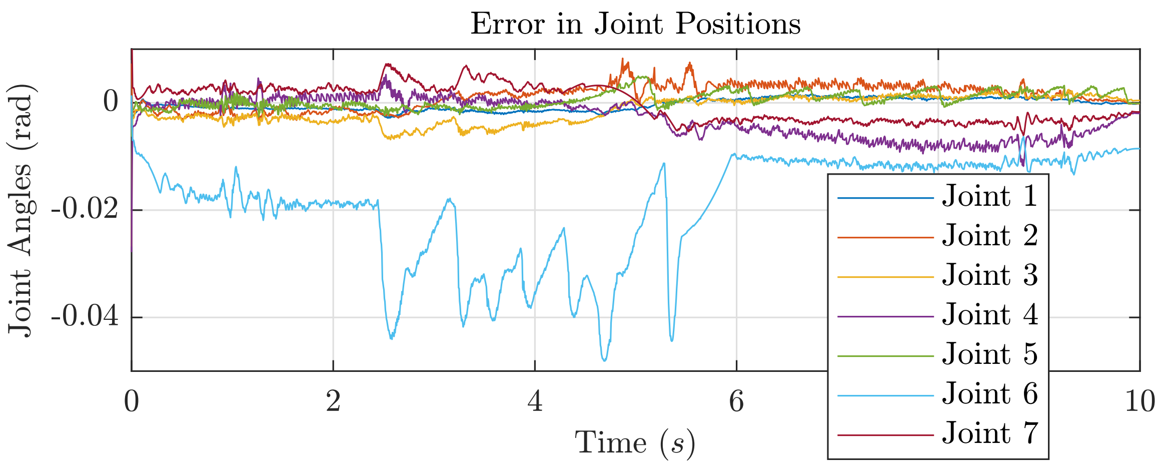

Figure 3.34 Absolute error in joint angle trajectories (experimental-simulated).

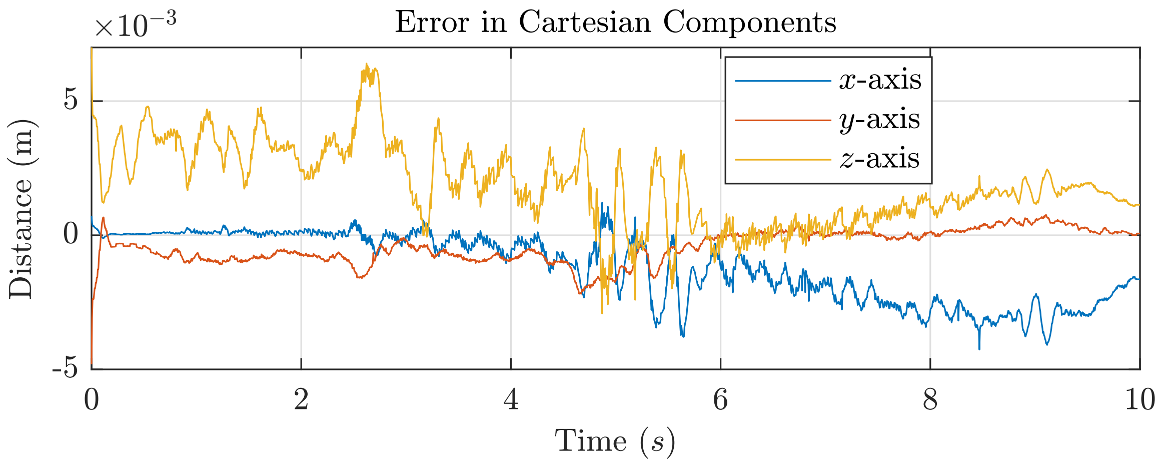

Figure 3.35 Absolute error in end-effector Cartesian coordinates.

3.4. Discussion

An open-loop simulation, straight line and curvilinear closed-loop simulations were completed. The closed-loop trajectories were implemented on the Sawyer robotic arm for validation purposes.

To ensure the accuracy of the analytical dynamic models (D-H parameters and PoE) the methods were compared to a numerical simulation developed in the SimScape^{TM} Multibody environment and experimental results obtained from the Sawyer robotic arm.

The comparison showed that the PoE calculates torque profiles in a different way as compared to the D-H and SimScape^{TM} approaches. The PoE approach was more accurate when compared against the experimental results, where the D-H and SimScape^{TM} methods were less accurate. These results demonstrate that the PoE approach is ultimately a more accurate approach for computing inverse dynamics.

A computed torque controller was used along with the Lie Group and Lie algebra representation of dynamics and kinematics of a 7-revolute joint open-chain manipulator to produce a straight-line motion of the end-effector in Cartesian space with quintic time-scaling that ensures finite jerk throughout its motion. This goal was achieved; however the jerk was not finite for some of the joints due to the fact that the finite-jerk motion constraint was only imposed on the motion of the end-effector in Cartesian space and not on the joint space. The controller demonstrated a degree of robustness and compensated for small uncertainties in the model parameters. Since a 7-revolute joint manipulator is a redundant system, the torque trajectories that correspond to the motion studied can be optimized by, for instance, penalizing the first rolling and pitching joints that move a lot of mass, which would theoretically decrease the amount of torque required for the same motion. The experimental results obtained by feeding the simulated joint space trajectories to the Sawyer robotic arm yielded accurate trajectories of the end-effector in Cartesian space despite the noise that was present in the recorded experimental joint angles, velocities and torques. In addition, the inverse dynamics algorithm's accuracy was validated by comparing the torque profiles of the simulated and experimental trajectories. The possibility of adding finite jerk constraints to the joint space while satisfying the constraints on the end-effector was explored and is discussed in Chapter 5.

4. Constant Screw Axis Trajectory Generation

In this chapter, the application of Neural Networks (NNs) and Machine Learning (ML) algorithms within the trajectory generation framework is presented. The main objective is to demonstrate that the trajectory of an end-effector of a multilink robotic system can be obtained by generating a path via the Bézier curve, imposing a finite jerk constraint, and using the NN and ML algorithms to obtain the Constant Screw Axis (CSA) trajectory closest to the chosen path. These constraints are of importance because the the finite jerk forces the jerk to be finite throughout the motion of an end-effector, which reduces the vibrations within the robotic arm or any other manipulator, and the CSA smooths the trajectory in Special Euclidean ($SE(3)$) space. First, a Bézier curve is chosen to define the path that an end-effector should follow. Subsequently, the finite jerk constraint is imposed. Lastly, NN and ML algorithms are used to obtain the closest CSA trajectory to the desired path. Since there are multiple CSA trajectories satisfying the initial configuration and final position, NN and ML algorithms are employed to minimize the Euclidean distance between the desired path (Bézier curve) and the actual obtained CSA trajectory.

4.1. Methodology

During the motion of a robotic arm or any other manipulator, an end-effector is tasked to move to a certain position at a specified time which is called a trajectory. The trajectory of an end-effector consists of a pure geometrical (Cartesian) position called the path and a time-scaling, which specifies the times when those geometrical (Cartesian) positions are reached (Lynch & Park, 2017).

There are numerous ways of defining both the path and the time-scaling of a trajectory. In this algorithm, Bézier curves are used to generate the path of an end-effector due to the virtue of the control polygon's ability to give a clear indication of whether the path is colliding with any nearby obstacle or object (Pastva, 1998). The convex hull points which describe a Bézier path are easily constrained in 3D space to ensure the arm does not collide with an obstacle in the path. A quintic polynomial time-scaling is chosen simply because it imposes the finite jerk constraint on the trajectory, which eliminates any undesired vibration of a robotic manipulator (Bobrow et al., 1985; Shin & McKay, 1985).

As the trajectory is fully defined, the Constant Screw Axis (CSA) trajectory is obtained by combining the knowledge of the initial and final Cartesian coordinates and the quintic time-scaling constraint. As there is no constraint on the final configuration's orientation, there are infinite CSA trajectories passing through the endpoints while satisfying the finite jerk constraint. However, as it is required to follow the path generated using Bézier curves, Machine Learning (ML) or Neural Networks (NN) can be used to minimize the distance between the desired generated path and the CSA trajectory by converging three parameters defining the endpoint's orientation.

4.1.1. Path and Time-Scaling

The control points for the path defined by Bézier curves lie within the work space of the manipulator, so that the end-effector can reach all cartesian positions given its geometrical limitation. An example of a Bézier curve with 4 control points is given in Figure 4.1.

The next step is to introduce the time-scaling that imposes the finite jerk constraint on the trajectory. The time-scaling that is capable of imposing this constraint is a quintic

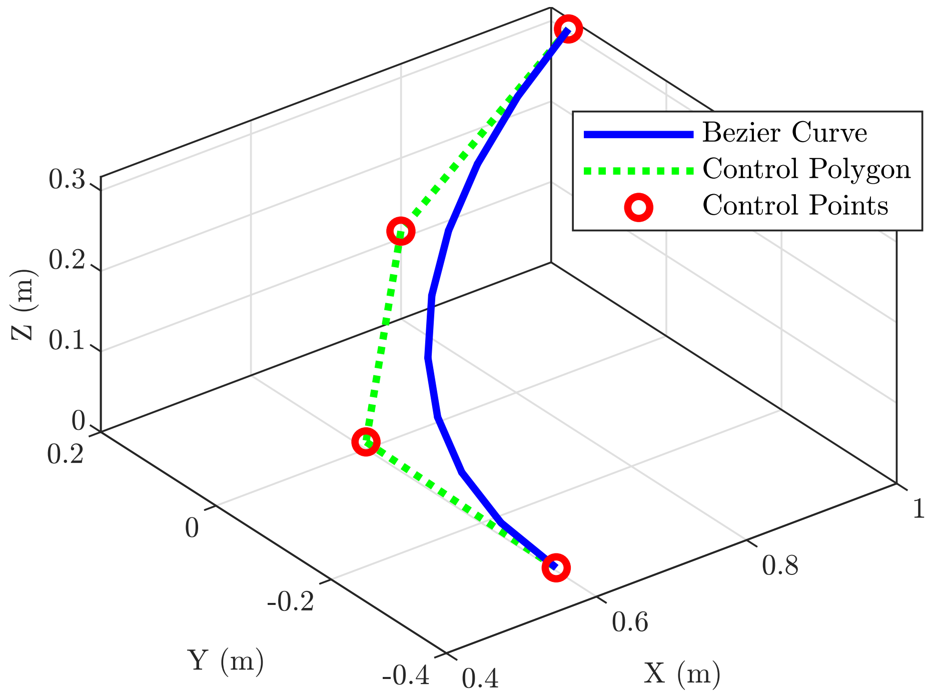

Figure 4.1 A Bézier curve example.

polynomial time-scaling function $s(t)$ that has the following mapping: $t \in [0, t_f]$, $s : [0, t_f] \rightarrow [0, 1]$. The time-scaling described in Equations 3.13-3.15 is utilized. Also, in order to avoid a discontinuous jump in acceleration, the path endpoints $\ddot{s}(0)$ and $\ddot{s}(t_f)$ are constrained to have zero acceleration, which forces the jerk to be finite throughout the whole path. The general expression for the quintic time-scaling is given by (Malik, Henderson, & Prazenica, 2021b):

$$s(t) = a_0 + a_1(t/t_f) + a_2(t/t_f)^2 + a_3(t/t_f)^3 + a_4(t/t_f)^4 + a_5(t/t_f)^5 \tag{4.1}$$

It should be noted that the time scaling represented in Figure 3.5 does not take into account the maximum allowable angular velocity or acceleration that the actuator is capable of producing. To resolve this, the t_f parameter is set to be large enough such that the torques required at the joints (inverse dynamics) do not exceed the actuator limits (Malik, Henderson, & Prazenica, 2021b).

4.1.2. Constant Screw Axis Parameters

As the path and the time-scaling are defined, the CSA trajectory can be obtained by solving the equation following (Lynch & Park, 2017):

$$T(s) = T_{\text{start}} \exp(\log(T_{\text{start}}^{-1} T_{\text{end}})s) \ , \ s \in [0, 1] \tag{4.2}$$

where, $T(s), T_{\text{start}}, T_{\text{end}} \in SE(3)$ are the configuration matrices of an end-effector at arbitrary initial and final times respectfully. Configuration matrices store the information regarding end-effector frame's (or any object's) orientation and position in the following form:

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} \in SE(3) \tag{4.3}$$

where $R \in SO(3)$ is the orientation of the end-effector with respect to the inertial frame represented as a member of the special orthogonal group and $p \in \mathbb{R}^3$ is a vector that represents the inertial position of the end-effector frame in Cartesian space. The initial configuration's T_{start} orientation and position are both defined, whereas the final configuration's T_{end} position is defined but the orientation is not. By manipulating the orientation of the final configuration R_{end}, the CSA trajectory's path changes, which can be employed to fit to the desired Bézier curve path. This can be accomplished by utilizing 3 rotation matrices in terms of 3 unknown angles ($\theta_1, \theta_2, \theta_3$):

$$R_{\text{end}} = R_{\text{start}} R_{\text{x}}(\theta_1) R_{\text{y}}(\theta_2) R_{\text{z}}(\theta_3) \tag{4.4}$$

where,

$$R_{\text{x}}(\theta_1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_1 & -\sin\theta_1 \\ 0 & \sin\theta_1 & \cos\theta_1 \end{bmatrix} \in SO(3) \tag{4.5}$$

$$R_{\mathrm{y}}(\theta_2) = \begin{bmatrix} \cos\theta_2 & 0 & \sin\theta_2 \\ 0 & 1 & 0 \\ -\sin\theta_2 & 0 & \cos\theta_2 \end{bmatrix} \in SO(3) \tag{4.6}$$

$$R_{\mathrm{z}}(\theta_3) = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \in SO(3) \tag{4.7}$$

Given the orientation R_{end} of the final configuration T_{end}, Equation 4.2 can be used to obtain the entire CSA trajectory. However, in order for the CSA trajectory's path to match with the desired Bézier curve path, the final orientation's parameters $(\theta_1, \theta_2, \theta_3)$ must converge to the matching values. This can be accomplished by an iterative optimization algorithm or with a NN (Malik, Lischuk, et al., 2021b).

This problem of computing matching parameters can be approached differently by investigating the Bézier curve path's first segment's relation to the known start frame (orientation) R_{start} and the last segment's relation to the final frame (orientation) R_{end}, such that the Bézier curve path starts and ends with the same "direction". In other words, determine how the start frame should be rotated such that, at the end of the path, the last segment has the same vector components in the last frame as the first segment's vector components in the start frame. The first segment of the Bézier curve in the start frame is given by:

$$\mathrm{v}^{(1)}_{\text{start}} = R^{-1}_{\text{start}} \mathrm{v}^{(1)}_{\mathrm{s}} \tag{4.8}$$

where $\mathrm{v}^{(1)}_{\mathrm{s}} \in \mathbb{R}^3$ is the vector components of the first segment of the path given in the inertial frame. The last segment represented in the final frame is given by:

$$\mathrm{v}^{(2)}_{\text{end}} = R^{-1}_{\text{end}} \mathrm{v}^{(2)}_{\mathrm{s}} \tag{4.9}$$

where $\mathrm{v}_{\mathrm{s}}^{(2)} \in \mathbb{R}^3$ is the last segment of the path given in the inertial frame. Now, in order for the CSA trajectory's path to be close to the Bézier curve path, vector components of the first segment in the start frame should be equal to vector components of the last segment given in the final frame:

$$\mathrm{v}_{\mathrm{start}}^{(1)} = \mathrm{v}_{\mathrm{end}}^{(2)} \tag{4.10}$$

This yields the following condition:

$$R_{\mathrm{start}}^{-1}\mathrm{v}_{\mathrm{s}}^{(1)} = R_{\mathrm{end}}^{-1}\mathrm{v}_{\mathrm{s}}^{(2)} \tag{4.11}$$

If the condition given in Equation 4.11 holds true, the orientation parameters $(\theta_1, \theta_2, \theta_3)$ have converged to the values that ensure that the "direction" of the Bézier curve is the same in both the starting and final orientations. However, it should be noted that Equation 4.11 is not a well posed problem and it has multiple solutions, all of which could be "close" to the Bézier curve but not fully matching, which also means that the CSA trajectory path is not guaranteed to be fully identical to the Bézier curve. Nevertheless, this approach helps with the mathematical formulation and provides a first guess in an iterative method or a NN.

4.1.3. Fitness Function

Alternatively, another method could be used to check the "closeness" of the CSA trajectory to the desired path defined by the Bézier curve. The following equation demonstrates the error of the actual CSA trajectory obtained by some arbitrary $\theta_1, \theta_2, \theta_3$ parameters, which is used to quantitatively evaluate the algorithm effectiveness:

$$E = \sqrt{(P_{\mathrm{a}} - P_{\mathrm{d}})^{\xi}[(P_{\mathrm{a}} - P_{\mathrm{d}})^{\xi}]^T} \tag{4.12}$$

where E is the scalar metric defining the magnitude of curve "fitness", P_{a} is an $\mathbb{R}^{3\times n}$ matrix that consists of "n" Cartesian points defining the CSA trajectory's path, P_{d} is an

$\mathbb{R}^{3\times n}$ matrix that consists of n Cartesian points defining the desired Bézier curve path, and ξ is a column-wise vector norm mapping such that: $P \in \mathbb{R}^{3\times n} \rightarrow p \in \mathbb{R}^{1\times n}$.

The selected NN and ML algorithms can use Equation 4.4 and then check the fitness of the CSA trajectory's path with the condition shown in Equation 4.11 or the fitness parameter given in Equation 4.12. The latter is used to quantitatively evaluate the effectiveness and compare the NN and ML algorithms.

4.2. Results

The PSO algorithm is initialized in this work by assigning particles to a search space (S-space) $S \in \mathbb{R}^3$. Initial particles are evenly spaced in the S-space with 5 coordinates ranging from $-90°$ to $+90°$ in all axes, which yields 125 equally spaced particles that are shown in Figure 4.2. Once particles x_i^k are initialized, the velocities v_i^k are initialized

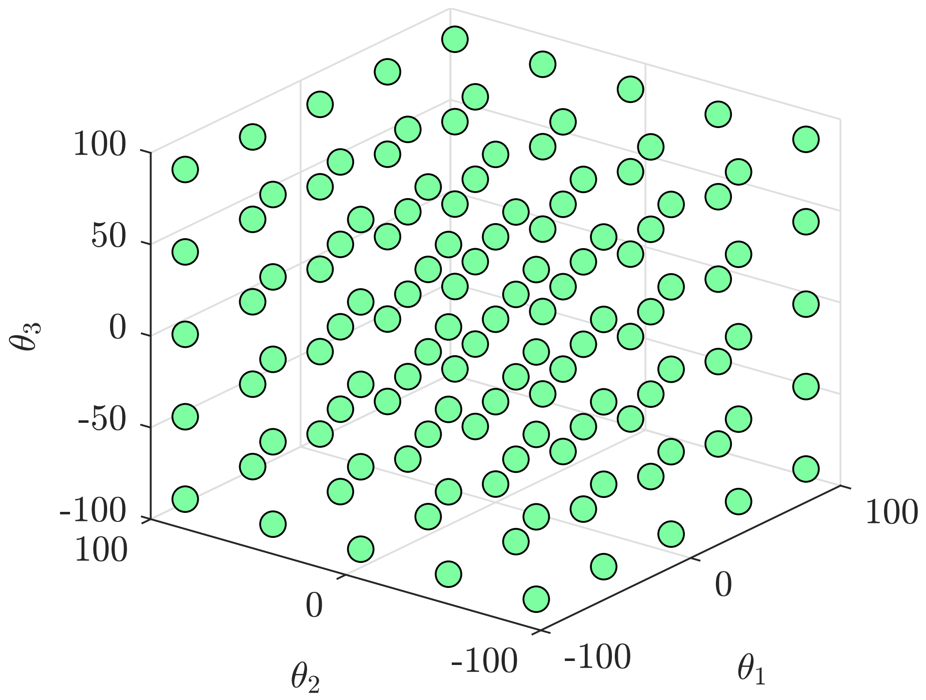

Figure 4.2 Particles x_i^0 assigned to S-space.

randomly. Each particle records the best personal fitness to the desired Bézier curve and is cognizant of the best global solution amongst the swarm. The fitness is checked by utilizing Equation 4.12. The velocity and position of each particle is updated according to the following update law:

$$v_i^{k+1} = \kappa[\omega v_i^k + c_1 r_1 (PB_i^k - x_i^k) + c_2 r_2 (GB^k - x_i^k)] \tag{4.13}$$

$$x_i^{k+1} = x_i^k + v_i^{k+1} \tag{4.14}$$

$$\kappa = \frac{2}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|}, \ \phi = c_1 + c_2 > 4 \tag{4.15}$$

where κ is the constriction factor limiting the magnitude of the velocity, ω is the inertia which controls the exploration and exploitation in the search space, c_1 and c_2 are cognitive and social parameters respectively, r_1 and r_2 are random variables with a range of $[0, 1]$, PB is the best recorded individual fitness, and GB is the best recorded global (swarm) fitness.

In order to assess the performance of the PSO algorithm, a sample Bézier curve trajectory was generated. The following parameters were used when applying PSO to fit the CSA trajectory to a Bézier curve: $\omega = 1$, $c_1 = 2$, $c_2 = 2.5$, r_1 and r_2 are random variables following a uniform distribution over the set $[0, 1]$. The stopping criteria were maximum number of iterations $k = 200$ or variance in the fitness of a given iteration, $\text{var}(E) < 1 \times 10^{-7}$. Starting from the initial particle distribution given in Figure 4.2 and using update Equations 4.13 - 4.15, the PSO algorithm was able to reach a minimal possible value of the fitness metric evaluated using Equation 4.12. Figure 4.3 demonstrates the convergence of the swarm in the S-space. Because of the stochastic parameters in the update law, the number of iterations required for convergence varied from run to run from approximately 16 to 20 iterations, and it should be noted that all 125 particles went through the provided number of iterations. It was observed that removing the constriction parameter κ from the swarm intelligence leads to an increased number of iterations that almost doubles in some cases.

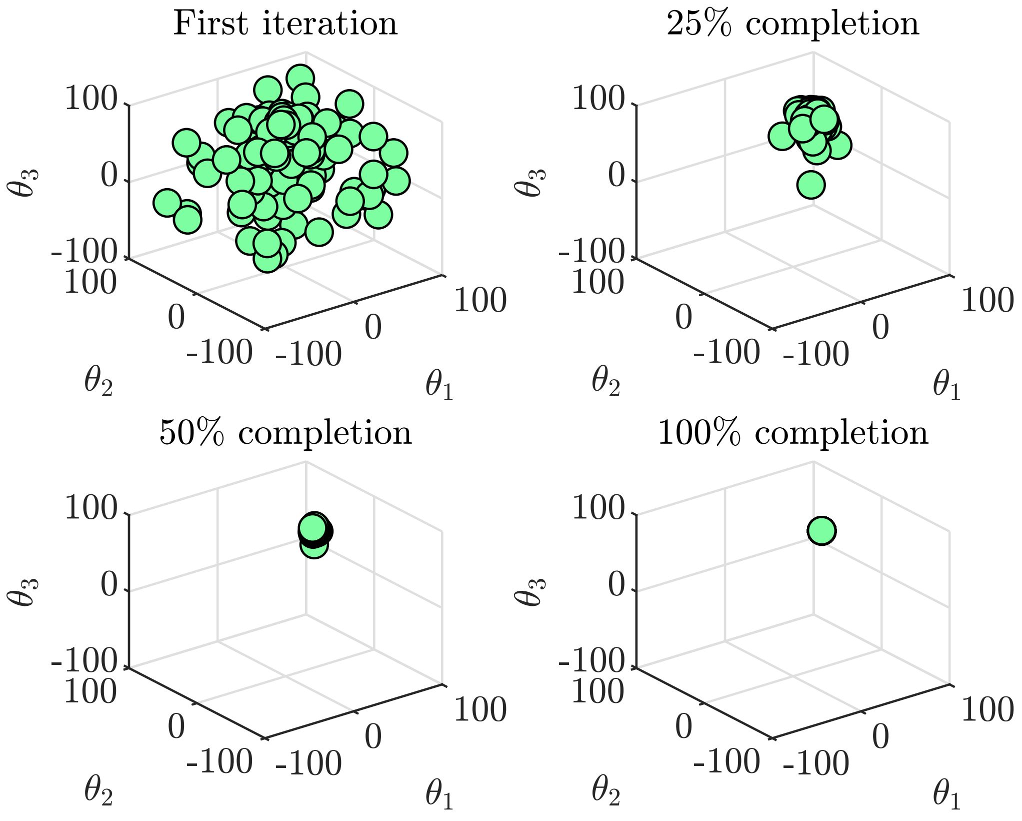

Figure 4.3 Particles movement in S-space as iterations progress.

Notably, Figure 4.4 demonstrates the comparison of the handpicked CSA trajectory (shown in blue), swarm CSA trajectory (shown in green), Q-Learning trajectory (shown in yellow), and DQN trajectory (shown in purple) with the desired Bézier curve path (shown in red). It can be seen that the CSA trajectories obtained from PSO, Q-Learning, and DQN are “closer” to the desired Bézier curve path; furthermore, these trajectories intersect with the desired path at some points and their average fitness values calculated using Equation 4.12 are 0.9478, 0.922, and 0.957 respectively, as shown in Table 4.2. In contrast, the handpicked trajectory does not cross paths with the desired trajectory and even has constant displacement error along the z-axis with fitness value of 0.985. However, it is important to mention that the swarm generated CSA trajectory still does not fully track the Bézier curve path, which can be attributed to the fact that the existence of a CSA path is not guaranteed for any given Bézier curve path. In other words, there might not be any axis along which the generated CSA trajectory can fully approximate the Bézier curve. In addition, the fitness metric was found to be case-sensitive as the value of the fitness highly depends on the chosen desired Bézier curve path. Also, since the computation using the PSO method is supposed to dynamically converge to a solution, a new calculation will need to be performed every time and for each new θ inputs.

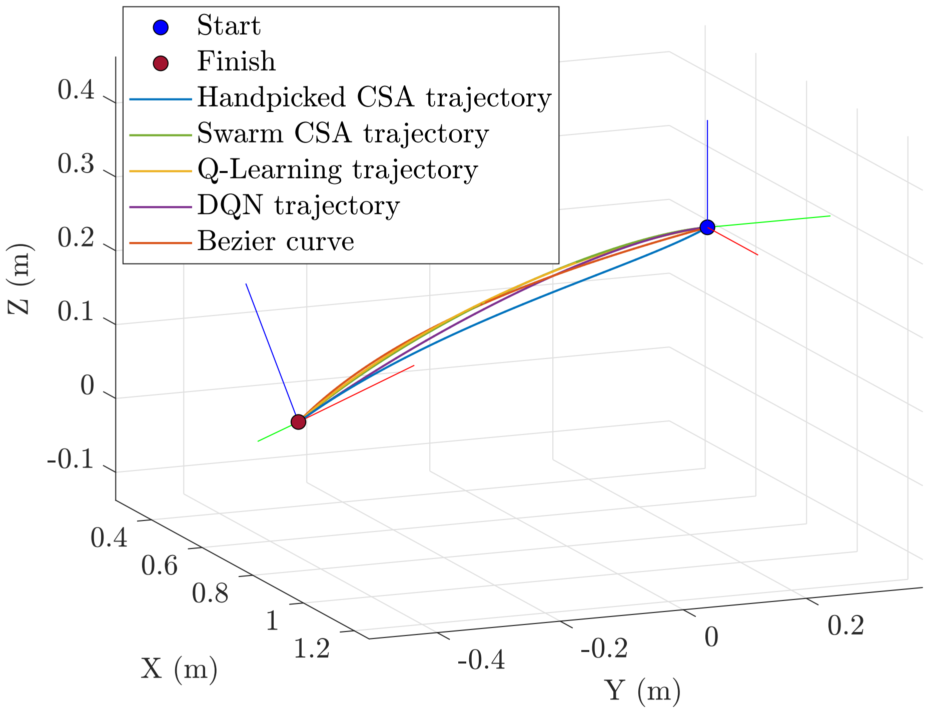

Figure 4.4 Comparison of CSA trajectories.

Q-Learning was also used to generate a CSA trajectory in order to compare with PSO, and with some handpicked θ inputs. The following parameters were used when applying Q-Learning to fit the CSA trajectory to a Bézier curve: the learning rate $\alpha = 1.1$, the discount factor $\gamma = 0.8$, the initial exploration factor $\epsilon = 1.0$, the initial θ vector was randomized with values from the set $[-180, 180]$, and the change per iteration in the θ vector is $\delta\theta = \pm 0.1\,\mathrm{deg}$. Three states were chosen: initial, transition, and final. Most calculations would occur during the transition state, where an action would be chosen, performed, evaluated and the reward would be given. It must be noted that having only three states allows to simplify computations and utilize Equation 4.12 for fitness calculation, similar to what was done for the PSO method. Three distinct types of actions were allowed: increase the angle, decrease the angle, or keep the angle unchanged. Furthermore, there are 3 elements in the θ vector, which results in a total of 27 items on the overall actions list.

To evaluate results after each iteration, the newly calculated fitness was compared to the fitness of the previous iteration and the reward was given based on the result. If fitness for the current iteration is lower than the fitness for the previous iteration, the agent will receive a positive reward. If the situation is the opposite or the current fitness equals the previous fitness, a negative reward will be given to the agent. A stopping criterion was added to the algorithm: the θ vector parameters are assumed to be converged if the variance in the fitness of 500 iterations is $\mathrm{var}(E) < 1 \times 10^{-8}$. It must be noted that the number of iterations required for the Q-Learning algorithm to converge highly depends on the initial value of the θ vector. However, it was found out that a maximum of 6000 iterations provides good convergence from any initial θ. An example of this convergence can be seen in Figure 4.5, where the calculation of the CSA error trajectory (fitness) according to Equation 4.12 is shown against the number of iterations. It can be seen that the algorithm converges rather quickly, yielding the CSA trajectory shown in Figure 4.4.

Despite providing a solution, however, this method suffers from a similar issue in terms of computed path storage. Although the Q-table provides the ability to store actions, it might not be very useful if the θ inputs change drastically. This is because, if the Q-table is stored permanently and reused with new inputs, the agent will attempt to bring itself to the same θ values that were computed during training. Depending on the starting θ, this may require long or short computation time. The main problem is that, for some θ, there might be closer, more applicable values than the ones computed by the Q-learning agent has solved for during training. Alternatively, depending on the θ inputs and if the Q-table is reused, the resultant CSA trajectory can have a very high fitness value, thus being quite far from the Bezier curve. To avoid these issues, similar to the PSO method, the Q-table can be recomputed each time for new θ inputs.

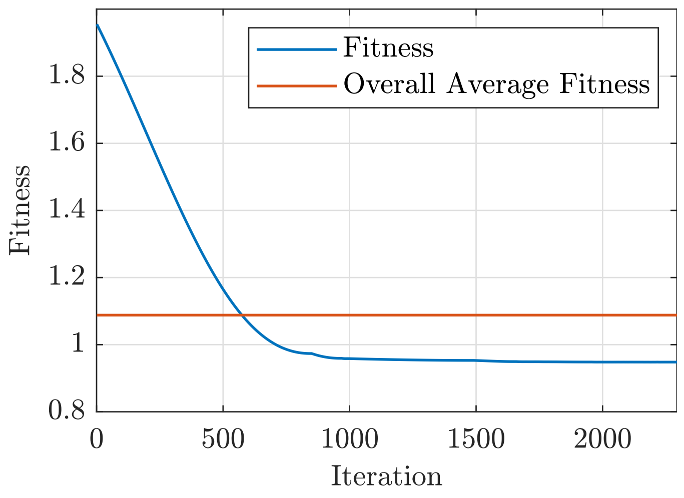

Figure 4.5 CSA error trajectory calculation by Q-Learning.

Finally, DQN was used to calculate the CSA trajectory. To implement this algorithm, MATLAB with its Deep Learning and Reinforcement Learning toolboxes was used. To comply with the MDP tuple, states were mapped to actions using a Neural Network. This network consisted of six layers: the input layer, four hidden layers, and an output layer. The first layer was selected to be a feature input layer with three nodes, each representing the value within the θ vector. Due to the fact that each θ can only be within the [-180, 180] limit, normalization was not chosen for this layer. The second layer was selected to be a fully connected layer with 24 nodes, followed by a ReLU layer that represents a Rectified Linear Unit activation function. Next is another fully connected layer with 48 nodes, which was followed by another ReLU layer. The number of nodes was chosen to simplify computations and, as a result, to decrease the time to perform it.

It was observed that increasing the number of nodes did not provide a significant improvement in computation of the θ values for the CSA trajectory. The last layer was chosen to be a fully connected layer with 27 nodes, where each node would represent an action. An evaluation of each state-action pair was performed similar to the PSO and Q-learning methods, using fitness calculation with Equation 4.12. A reward was given to the DQN agent in a similar fashion as for the Q-learning method. That is, the agent is rewarded if the fitness for the current pass is lower than its value for the previous pass. The terminal condition was determined using computation of the variance in the fitness of 500 iterations for each training epoch, which corresponded to $\operatorname{var}(E) < 1 \times 10^{-8}$. Other training parameters for the NN were set as shown in Table 4.1.

Similarly to the PSO and Q-learning methods, DQN converges to a solution where the fitness is minimized, as can be seen in Figure 4.4. It was also observed that, at times,

Table 4.1

Selected DQN Parameters

θ vector	between [-180, 180]
Change per iteration $\delta\theta$	$\pm 0.1\,\text{deg}$
Learning rate	0.1
Discount factor	0.9
Gradient Threshold	1
Mini Batch Size	64
Target Smooth Factor	1e-3

DQN can find a solution that has lower fitness than Q-learning, although it takes DQN longer to do so due to its complexity. This can be mitigated by increasing the learning rate to about 0.7, allowing DQN to find the optimal solution quicker without much overshooting. A comparison of PSO to Q-learning and DQN using various parameters with handpicked θ vectors is provided in Table 4.2. It must be noted that an average was taken over four cases for different θ inputs, where finding the CSA trajectory for each case was done for five trials.

Table 4.2

Algorithm comparison

Property	PSO	Q-learning	DQN
Avg fitness	0.948	0.922	0.957
Avg computation time	6.00	9.56	88.06
Avg iterations to converge	2125*	3344	4328

Despite being able to converge to a solution with minimized fitness, DQN was found to have issues with the overall stability. This is because DQN performs better with discrete action spaces, rather than with continuous ones. For our problem, as was mentioned earlier, the action can be either increasing the angle by a constant increment $\delta\theta$, decreasing the angle by a constant $\delta\theta$, or keeping the angle unchanged. However, this property can also be continuous, meaning that it can change after each iteration of the calculation. For example, in one iteration, an angle can be chosen by an algorithm to be decreased by 0.5 degrees, but in the next iteration it can be decreased by 0.1 degrees, suggesting that the reinforcement learning agent should understand that it does not have to increment the angle by too much to avoid overshooting. Having discrete angle increments works well for this problem when PSO and Q-learning are used, as these algorithms only need to converge to one solution and training through multiple epochs is not required, i.e. computation can occur in one epoch with multiple iterations.

Additionally, PSO and Q-learning are less complex than DQN; “remembering” solutions is not required and a computation for new inputs can be done rather quickly. A DQN agent, on the other hand, is intended to be trained, saved and reused. However, with the large solution space that the Bézier curve trajectory calculation poses, not every action will be the same for different inputs; thus, for different θ vectors, different actions are required to converge to a trajectory that will be reasonably close to a Bézier curve. This can perhaps be mitigated by training the DQN agent for a large number of epochs, possibly 10000 epochs or more. Another solution can be to implement an actor-critic strategy for the algorithm, where the critic will be more active and responsible for evaluating state observations and actions using Neural Networks, and the actor will be a separate Neural Network that will evaluate state observations for maximizing the long-term reward. This can be done for DQN, however, another algorithm such as Deep Deterministic Policy Gradient (DDPG) may be more applicable, as it is intended to work with continuous action spaces (Casas, 2017). For CSA trajectory calculation using this method, a continuous action space can be represented as changing the value of each θ by a non-constant number

$\delta\theta$ that will be updated every iteration and will lie within a certain range, for example between [-10, 10] degrees. This is in contrast to the discrete action space that was used for PSO, Q-Learning and DQN, where each θ was incremented/decremented each time by 0.1 deg.

4.3. Discussion

It was demonstrated that PSO, Q-leaning and DQN algorithms are able to generate CSA trajectories “close” to the desired Bézier curve path, based on the fitness metric, in a reasonable number of iterations. The algorithm can be generalized to any degree of freedom robotic manipulator provided that the inverse kinematics solution satisfying the end-effector path generated by the algorithm exists. Even though the goal of the algorithms was to minimize the fitness value, it was observed that the existence of the CSA trajectory that corresponds to a desired Bézier curve path is not guaranteed, which makes the fitness parameter case-sensitive and not always applicable.

Nevertheless, the algorithms were able to find the “closest” curves for a given case within a reasonable computation time and number of iterations. The PSO algorithm was found to be the fastest, as was expected, with an average computation time of 6 seconds, and also the most consistent, as fitness values for different trials within the same case study were exactly the same, even though initial velocities were always randomized for each trial.

Q-learning produced better fitness values on average; however, it was less consistent and a bit slower than PSO, having the mean time of 9.56 seconds. In some cases, the fitness value varied more than PSO for the same inputs and throughout different trials. This indicates that PSO is more stable on average, and given its computation time, it can be argued that it can be used for near real-time operation, despite having to perform a new computation each time new inputs are introduced.

DQN was also able to find the “closest” curves; however, it posed certain challenges to be reliably reused as a pre-trained agent for any initial CSA trajectory θ input. To accomplish this, it is possible that a lengthy and time consuming training will be required. This training must encompass finding “closest” curves for a least 10000 epochs. On the contrary, an alternative solution can be to utilize the DDPG algorithm. It works similar to DQN but is more aligned to be used with continuous action spaces, as in the curve calculation described in this chapter.

5. Swarm Intelligence Trajectory Generation

This chapter is aimed to demonstrate a multi-objective joint trajectory generation algorithm for a 7 Degree of Freedom (DoF) robotic manipulator using a combined Swarm Intelligence (SI) - Product of Exponentials (PoE) method. Given a priori knowledge of the end-effector Cartesian trajectory and obstacles in the workspace, the inverse kinematics problem is addressed usin SI-PoE subject to multiple constraints. The algorithm is designed to satisfy a finite jerk constraint on the end-effector, avoid obstacles, and minimize control effort while tracking the Cartesian trajectory. The SI-POE algorithm is compared with conventional inverse kinematics algorithms and standard Particle Swarm Optimization (PSO). The joint trajectories produced by SI-POE are experimentally tested on the Sawyer 7 DoF robotic arm, and the resulting torque trajectories are compared.

The rest of the chapter is outlined as follows: Section 5.1. describes the robotic arm, finite jerk end-effector trajectory generation, and the PoE forward kinematics (PoE-FK) algorithm used in setting up the IK problem and fitness function for SI. Section 5.2. outlines the SI-PoE algorithm and its fitness function along with parameters, collision detection mechanism, particles' initial conditions, and computational performance. Section 5.3. presents the resulting joint trajectories, methods used to smooth them, and error in the end-effector's position. Section 5.4. provides the simulated and experimental torque profiles and discussion on how the SI-PoE parameteres can be varied case-by-case for optimal performance. Finally, Section 5.5. will provide discussion and final remarks.

5.1. Methodology

The robotic arm used in the simulation and experiments is Rethink Robotics' 7-revolute (7R) Sawyer robot, which is shown in Figure 1.1, where the robotic arm is at its home configuration with all joint positions at zero ($\theta_1 = 0, \theta_2 = 0, \cdots, \theta_7 = 0$). The home configuration and all of the dimensions were taken from Universal Robot Description Format (URDF) file dedicated to the Sawyer robot Robotics (2017), and the simple geometry reconstructed in Matlab is shown in Figure 5.1. The Sawyer arm has 7 links and 7 revolute joints, 4 of which are rolling and 3 are pitching joints.

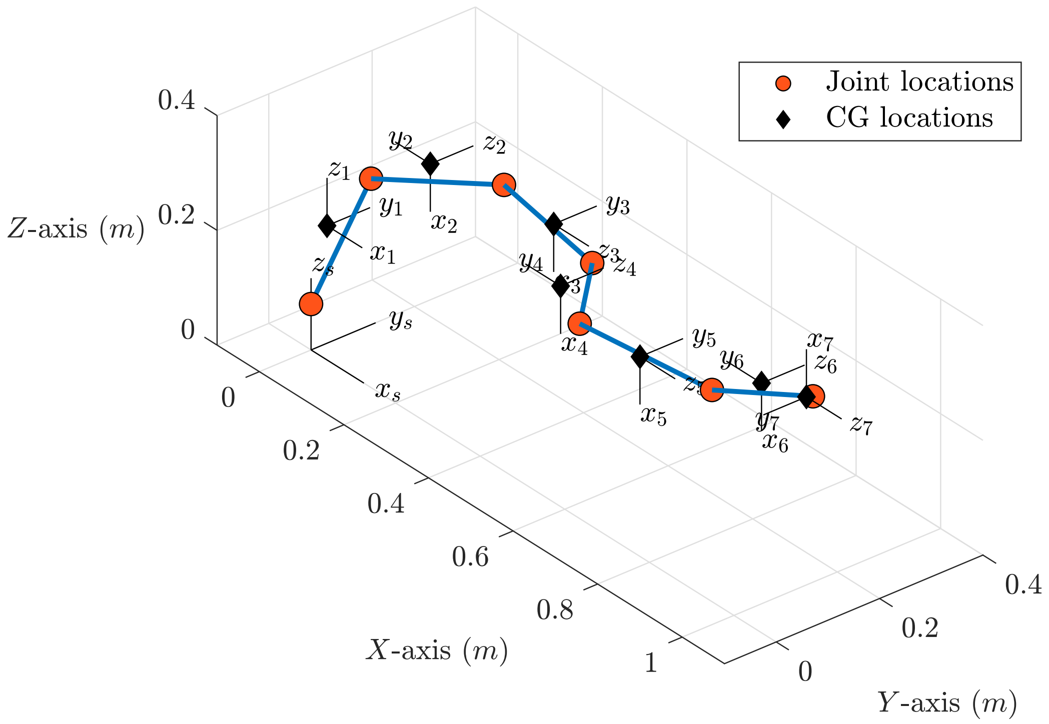

Figure 5.1 Home configuration of the Sawyer robotic arm represented in *Matlab*.

Forward kinematics is realized using PoE-FK. This method was chosen because it enjoys certain advantages over the conventional Denavit-Hartenberg parameters (D-H) approach, which include but are not limited to intuitive geometric interpretation that leads to an easier set up process, uniform treatment of revolute and prismatic joints, absence of strict rules to assign frames, and concise and elegant formula.

Given physical locations of joints in home configuration from the URDF file, the screw axes in the space frame are shown in Equation 1.13. The PoE-FK formula, which represents the position and orientation of a frame (point) attached to the n-th link, is shown below:

$$T_{sn} = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \cdots e^{[\mathcal{S}_n]\theta_n} M_{sn} \tag{5.1}$$

where $M_{sn} \in SE(3)$ is a frame (position and orientation) attached to the robotic arm's n-th link given in home configuration. The complete derivation of the PoE-FK can be found in Chapter 1. Now, the choice of frames (points) to be tracked by PoE-FK is of paramount importance and is chosen by the user (Malik, Henderson, & Prazenica, 2021a). For example, if the center of gravity (CG) of each link is to be tracked, the following matrices extracted from the URDF file can be used:

$$M_{s1} = \begin{bmatrix} 1 & 0 & 0 & 0.0244 \\ 0 & 1 & 0 & 0.0110 \\ 0 & 0 & 1 & 0.2236 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{s2} = \begin{bmatrix} 0 & -1 & 0 & 0.1078 \\ 0 & 0 & 1 & 0.1425 \\ -1 & 0 & 0 & 0.3201 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

$$M_{s3} = \begin{bmatrix} 0 & 0 & 1 & 0.3568 \\ 0 & 1 & 0 & 0.1775 \\ -1 & 0 & 0 & 0.3172 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{s4} = \begin{bmatrix} 0 & -1 & 1 & 0.5091 \\ 0 & 0 & 1 & 0.0663 \\ -1 & 0 & 0 & 0.3218 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

$$M_{s5} = \begin{bmatrix} 0 & 0 & 1 & 0.7401 \\ 0 & 1 & 0 & 0.0309 \\ -1 & 0 & 0 & 0.3189 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{s6} = \begin{bmatrix} 0 & -1 & 0 & 0.9047 \\ 0 & 0 & 1 & 0.1314 \\ -1 & 0 & 0 & 0.3109 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

$$M_{s7} = \begin{bmatrix} 0 & 0 & 1 & 0.9860 \\ 0 & -1 & 0 & 0.1517 \\ 1 & 0 & 0 & 0.3170 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

However, it is important to mention that the computational effort increases per increment of the number of points calculated by PoE-FK. The SI-PoE algorithm uses these points calculated by PoE-FK to detect collisions, which could pose certain problems if few points on the robotic arm are tracked. For instance, if few points are checked for collision, a small obstacle might be passing between these two points without collision. However, the physical links might be colliding with that obstacle.

Thus, the size of obstacles in the workspace is an important consideration, since both the computational effort and the collision detection accuracy of the SI-PoE directly correlate to the number of virtual points on the robotic arm that are chosen to detect collisions and avoid obstacles. If few number of virtual points are implemented, the boundary surrounding the obstacle can be increased in size to reflect collision. But this approach will constrain the joint movement even further, which would limit the search space for the SI-PoE algorithm, leading to underutilized collision-free space, undesirable joint trajectories, or even absence of a solution in critical cases. In this case, a good approach would be to choose, for instance, the CGs of all links as frames (points) for tracking, interpolate between them depending on the heuristic of relative obstacle sizes, and choose the appropriate size of boundaries surrounding obstacles. This way, the PoE-FK (Equation 5.1) has to be called only for the CGs and not for virtual points between them, which considerably decreases the computational effort while maximizing the collision-free space.

The trajectory of the end-effector is generated by utilizing Bézier curves for path generation, and finite jerk model for the time-scaling (Malik, Henderson, & Prazenica, 2021b). In order to satisfy the finite jerk constraint on the end-effector, a quintic polynomial time-scaling presented in Equations 3.13 through 3.15 is employed. The plots of the time-scaling used to obtain the finite jerk trajectory is presented in Figure 3.5. The finite time t_f is chosen to be large enough so that the Sawyer robotic arm's joint rates are not saturated to the maximum.

An example of a Bézier end-effector path is shown in Figure 5.2. By applying the time-scaling given in Equations 3.13 through 3.15, the trajectory now can be fully defined. The resulting trajectory is smooth and satisfies the finite jerk constraint throughout the whole duration of the movement. However, it should be noted that only the end-effector movement satisfies the finite jerk constraint. Applying similar constraints on joint movement is shown in Section 5.3.

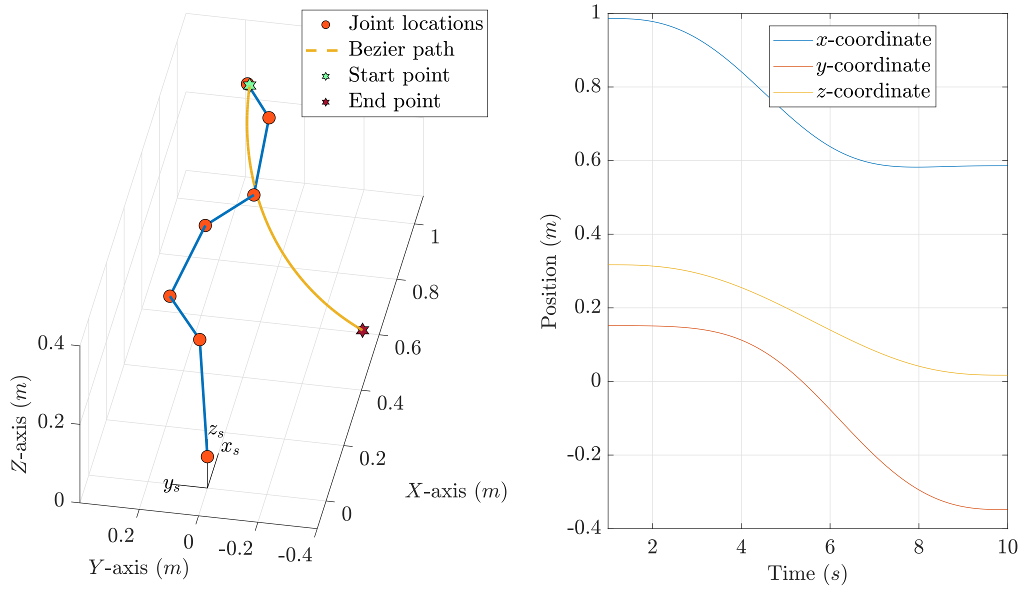

Figure 5.2 A sample trajectory generated using Bézier path.

The trajectory shown in Figure 5.2 was used as an input example to both the SI-PoE algorithm simulation and the SI-PoE Sawyer experiment. The path of the end-effector was generated using a Bézier curve, and quintic time-scaling given in Equation 3.15 was used to produce the finite jerk profile. The objective of the SI-PoE simulation is to check the algorithm's accuracy and efficiency; and the objective of the experiments is to validate that the joint trajectories produced by SI-PoE can be followed accurately by the Sawyer robotic arm. The experimental torque profile obtained in the Sawyer experiments is compared with the PoE inverse dynamics formulation for the Sawyer robot developed in Chapter 3.

5.2. SI-PoE Algorithm

The core of the SI-PoE algorithm is PSO and PoE. The latter was described in Section 5.1., while the former is a method that is a part of a larger family of SI algorithms, which describe social behavior of various ecosystems and animals such as bird flocks, schools of fish, etc (Eberhart & Kennedy, 1995; X.-S. Yang et al., 2013). PSO is metaheuristic by nature, straightforward in implementation, and converges relatively quickly (Y. Shi & Eberhart, 1999). These are the main reasons why it is utilized in a variety of disciplines and applications. In simple terms, PSO iteratively searches through the solution space using particles. Each particle contains parameters representing a solution (fitness), that denotes its current position in the given solution/search space, as well as velocity, which influences its position (fitness), guiding it to the most optimal solution. In general, PSO is a global optimization algorithm, and therefore can provide solutions within a large search space, which is very attractive for application to high-DoF robotic manipulators. However, PSO suffers from high computational effort when solutions with a large degree of accuracy are required. Thus, combining PSO with a relatively fast PoE-FK algorithm is proposed in this work. Furthermore, an additional advantage of combining PSO and PoE is the ease of implementation of multiple objectives due to the concise and elegant forward kinematics implementation. The SI-PoE method can be easily generalized to any n-DoF robotic manipulator.

5.2.1. Swarm Initialization

The SI-PoE algorithm is initialized by assigning particles $\underline{x}_i \in \mathbb{R}^7$ to a search space (S-space) $S_{space} \in \mathbb{R}^7$. The initial location of the particles can be assigned arbitrarily, or in a specific fashion that is proposed in this algorithm. In the proposed SI-PoE particle

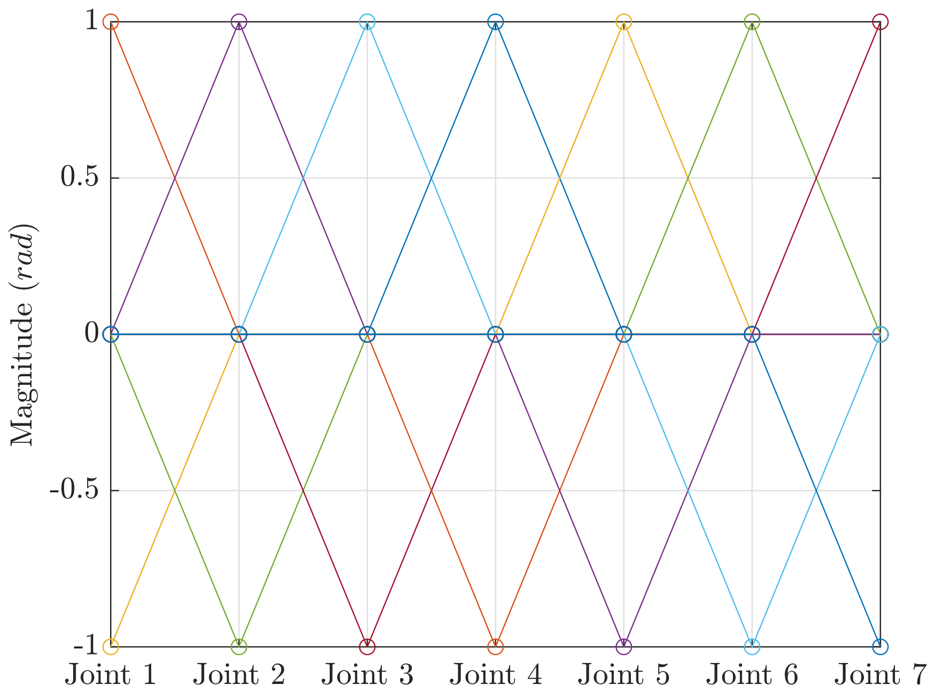

Figure 5.3 Particles $\underline{x}_i^0$ assigned to S-space starting from home configuration.

assignment, particles are evenly spaced in the S-space with an equal offset of $\pm 1 rad$ from the "previous" solution in each (joint) coordinate, such that the swarm consists of 15 points as demonstrated in Figure 5.3, where each 7-dimensional swarm particle is a line.

This method of particles' initialization works remarkably well due to the fact that each swarm point is essentially a change only in one of the 7 joint positions.

Most works choose the random method of swarm particles' initialization. However, when solving the IK problem, the known previous joint positions can serve as an initial condition or basis for the particles' distribution for each time step, which speeds up the search process while not limiting S-space. The IK computation time comparing random and the proposed particles' initial conditions is demonstrated in Table 5.1, where both a single end-effector position and a trajectory IK solution times are shown. The random particle's initialization was realized by adding random offsets in the range of $[-0.01, 0.01]\,rad$ to the previous successful joint positions. The stopping criteria were number of iterations $(k \leq 20)$, and fitness to the desired trajectory $(f < 0.0005)$.

Table 5.1

Comparison of random and proposed particles' initialization

Method	1 point IK (s)	Trajectory IK (s)
Random	0.24	13.61
Proposed	0.21	12.85

Although, the computational time advantage of the proposed method may seem marginal, the real advantage becomes evident when comparing the consistency of the two methods. The computation times demonstrated in Table 5.1 reflect the maximum number of iterations since both methods were not able to meet the fitness criterion. However, the proposed method was able to generate a trajectory with better accuracy in the same number of iterations as the random method. This is demonstrated in Figure 5.4, where the proposed swarm assignment has consistently small error throughout the trajectory, and the random swarm, on the other hand, exhibits large spikes in position error. It should be noted that both methods enjoy the advantage of having previously calculated joint positions, which drastically reduces the computation time and the scope of search in S-space.

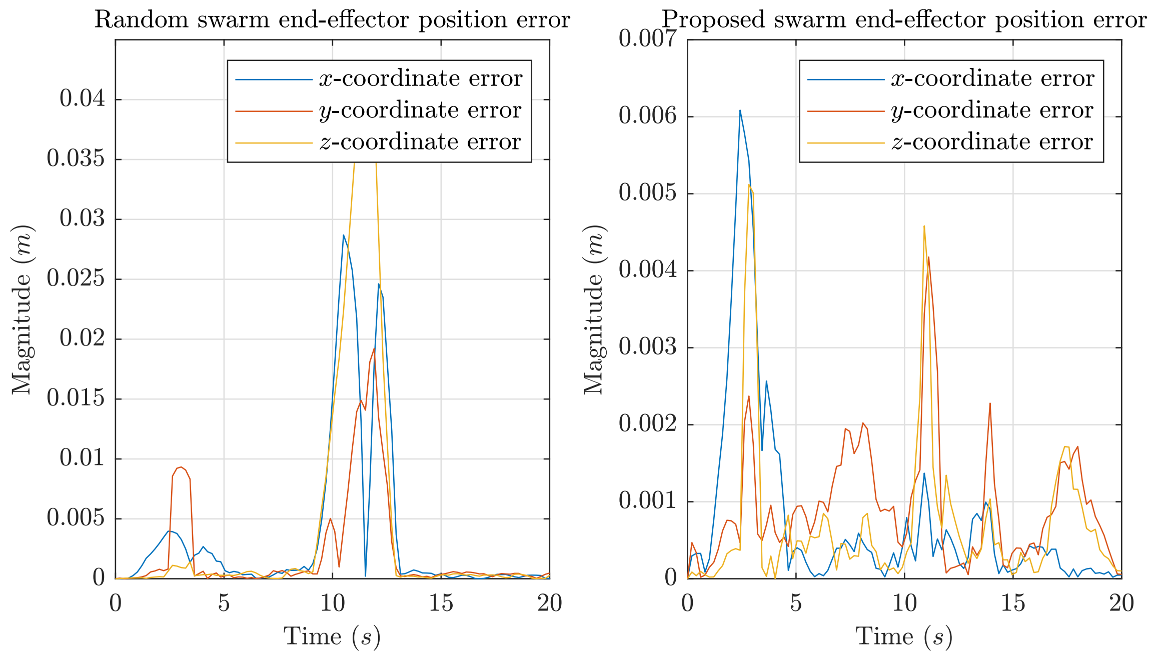

Figure 5.4 Position error with random swarm assignment (left) and proposed (right).

5.2.2. Swarm Fitness Function

As the objectives of the SI-PoE algorithm are to track pre-assigned trajectory while satisfying finite jerk, minimizing joint effort, and avoiding obstacles, the fitness function is shown below, where both the orientation and position of the end-effector can be tracked:

$$f_i = \sigma_p \| \underline{p}_i - \underline{p}_d \| + \sigma_R \arccos\left(\frac{tr(R_i R_d^T) - 1}{2} \right) + \underline{\sigma}_J^T \mid \underline{x}_i^k - \underline{\theta}_p \mid_e + coll \tag{5.6}$$

where $R_i \in SO(3)$ and $\underline{p}_i \in \mathbb{R}^3$ are computed from PoE-FK,

$$T_{si} = e^{[\mathcal{S}_1]x_i^k(1)} e^{[\mathcal{S}_2]x_i^k(2)} \cdots e^{[\mathcal{S}_n]x_i^k(7)} M_{se} = \begin{bmatrix} R_i & \underline{p}_i \\ 0 & 1 \end{bmatrix} \in SE(3) \tag{5.7}$$

$$M_{se} = \begin{bmatrix} 0 & 0 & 1 & 0.9860 \\ 0 & -1 & 0 & 0.1517 \\ 1 & 0 & 0 & 0.3170 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.8}$$

The σ_p and σ_R are the weighting parameters for position and orientation respectively. Either of them can be set to zero for the cases where only position or attitude tracking is desired; $\underline{\sigma}_J = [0.01\ 0.009\ 0.008\ 0.007\ 0.006\ 0.005\ 0.004]^T$ is the parameter penalizing excessive joint movement, where the joints closest to the base are prioritized as they move more mass; $|\cdot|_e$ is the element-wise absolute value operator; M_{se} is the end-effector's home configuration; $\underline{x}_i^k$ is the i-th particle in k-th iteration, $\underline{\theta}_p$ is the current joint angles (successful previous iteration), and $coll$ is a scalar representing if any of the virtual points collide with an obstacle in the workspace if the robot assumes the $\underline{x}_i^k$ swarm particle's posture.

5.2.3. Collision Detection and Swarm Update Law

Collisions are detected using PoE, where the number of virtual points (frames) checked by the SI-PoE depends on the obstacles' size. The virtual particles are attached to a specified link and this dictates their PoE-FK formula. For an arbitrary j-th particle that is attached to the m-th link, the PoE-FK can be demonstrated as:

$$T_{sj} = e^{[\mathcal{S}_1]x_i^k(1)} \cdots e^{[\mathcal{S}_m]x_i^k(m)} M_{sj} \tag{5.9}$$

where M_{sj} is the position and orientation of the virtual particle in home configuration given in the inertial frame.

$$coll = \begin{cases} +1, & \text{if virtual point collides with an obstacle} \\ \\ 0, & \text{if virtual point does not collide with an obstacle} \end{cases} \tag{5.10}$$

After the $\underline{x}_i^0 \in \mathbb{R}^7$ particles are initialized, the velocities of the particles $\underline{v}_i^0 \in \mathbb{R}^7$ are randomly initialized with values from the $[-0.1, 0.1]$ range. The SI-PoE update law for the velocities and particles is presented in Equation 5.11 and 5.12, respectively. Numerical values of the update law hyperparameters are shown in Table 5.2.

$$\underline{v}_i^{k+1} = \kappa[\omega \underline{v}_i^k + c_1 r_1 (PB_i^k - \underline{x}_i^k) + c_2 r_2 (GB^k - \underline{x}_i^k)] \tag{5.11}$$

$$\underline{x}_i^{k+1} = \underline{x}_i^k + \underline{v}_i^{k+1} \tag{5.12}$$

$$\kappa = \frac{2}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|}, \ \phi = c_1 + c_2 > 4 \tag{5.13}$$

where κ is the constriction factor limiting the magnitude of particles' velocity, ω is the inertia weight which controls the exploration and exploitation in the search space, c_1 and c_2 are cognitive and social parameters respectively, r_1 and r_2 are random variables with a range of $[0, 1]$, PB is the best recorded individual particle's location in S-space $(\mathbb{R}^7)$, and GB is the best recorded global (swarm) particle location in S-space $(\mathbb{R}^7)$. The SI-PoE algorithm stops if either 20 iterations were attempted or if the fitness of the current global best particle is less than the set parameter ($f_{GB} < 0.0005$). The SI-PoE algoritm steps are summarized in the flowchart shown in Figure 5.6.

Figure 5.5 demonstrates an example of the swarm convergence in the S-space. The SI-PoE swarm consists of 15 points in S-space ($\mathbb{R}^7$), and it can be seen that the swarm almost converged at 50% completion. Because of the stochastic nature of the parameters within the update law, the number of iterations untill convergence varied from run to run from approximately 17 to 20 iterations. However, most of the time for the cases with $\sigma_R = 0$, the SI-PoE was able to find an accurate posture of the robotic arm within $10 - 12$ iterations with the position error of the end-effector being less than $8 - 9\,mm$, and the rest of the iterations of the algorithm reduces the position error to around $5 - 6\,mm$. It was observed that the constriction parameter κ reduces the number of iterations required to achieve said accuracy, thus positively contributing to the algorithm by decreasing the computation time.

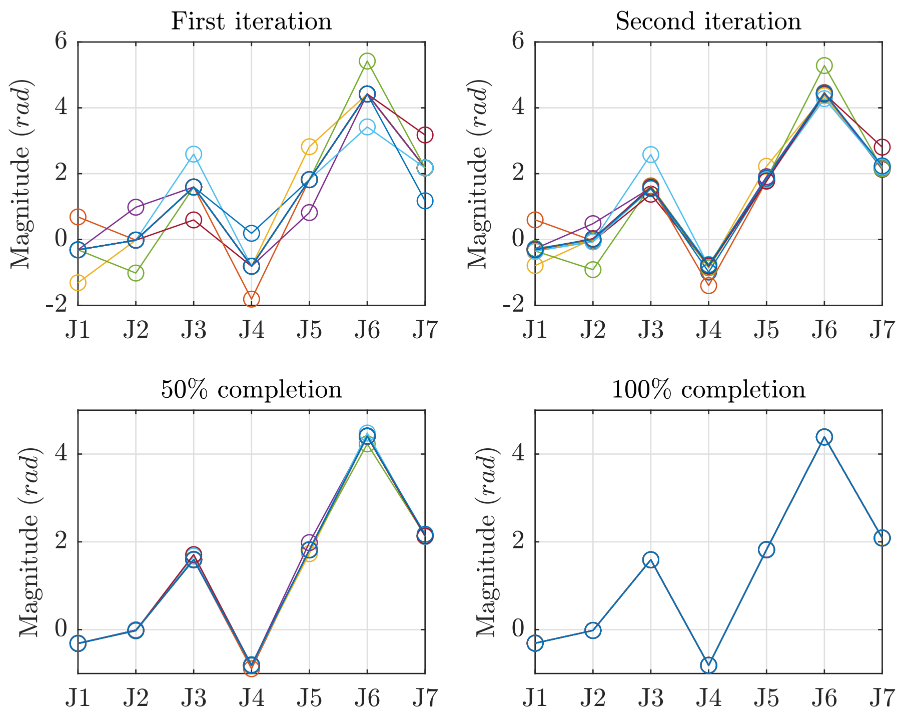

Figure 5.5 Evolution of the SI-PoE swarm.

Table 5.2

Selected SI-PoE parameters

Description	Variable	Value
Initial swarm point	$\underline{x}_i^0$	1 rad offset at each joint
Initial velocity	$\underline{v}_i^0$	random values between $[-0.1, 0.1]$
Inertia weight	ω	1
Cognitive parameter	c_1	2
Social parameter	c_2	2.5
Random variable	r_1	random value between $[0, 1]$
Random variable	r_2	random value between $[0, 1]$
Convergence parameter	ϕ	4.5
Constriction factor	κ	0.5

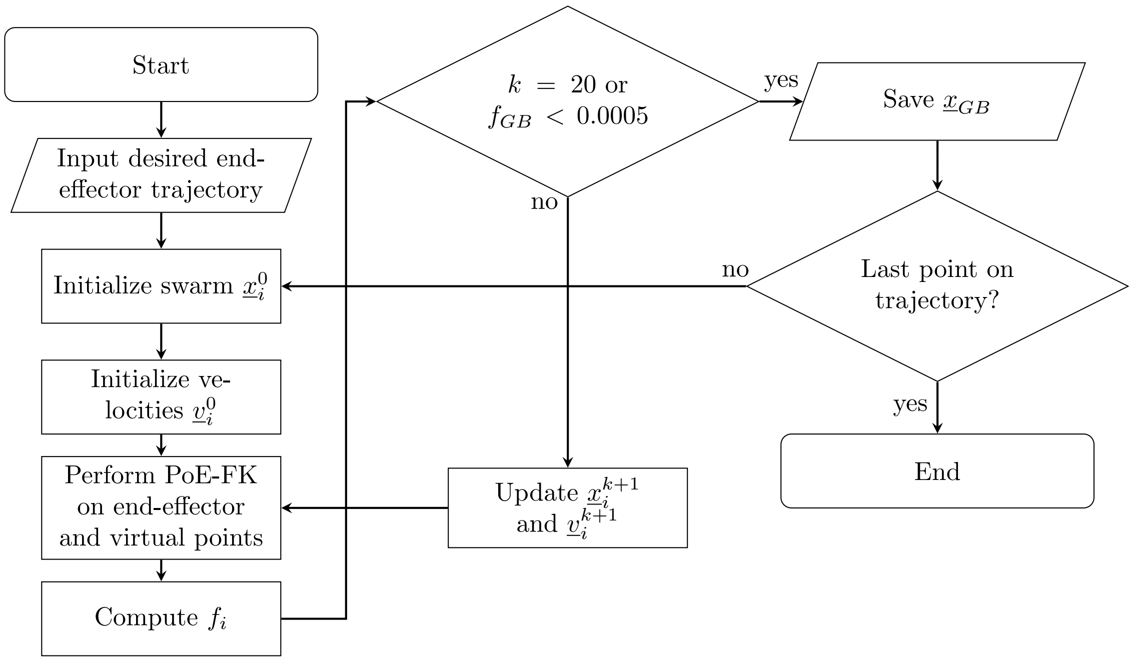
Start
Input desired end-effector trajectory
Initialize swarm $\underline{x}_i^0$
Initialize velocities $\underline{v}_i^0$
Perform PoE-FK on end-effector and virtual points
Compute f_i
$k = 20$ or $f_{GB} < 0.0005$
yes
no
Save $\underline{x}_{GB}$
Last point on trajectory?
no
yes
End
Update $\underline{x}_i^{k+1}$ and $\underline{v}_i^{k+1}$

Figure 5.6 SI-PoE algorithm flowchart.

5.3. Results

The SI-PoE algorithm was tested on different trajectories. For the purpose of illustration and without the loss of generality, the end-effector trajectory shown in Figure 5.2 served as an input to the SI-PoE algorithm, and 3 obstacles were added to the workspace. All of the obstacles were spheres: two of them had a diameter of $20\ cm$ centered at $[0.5, 0, 0.35]^T$ and $[0.6, -0.13, 0.4]^T$, and the third obstacle had a diameter of $40\ \ cm$ with its center at $[0.5, 0, 0]^T$. The center of gravity of each link served as a virtual point for obstacle detection, as outlined in Equation 6.2, resulting in a total of 8 virtual points.

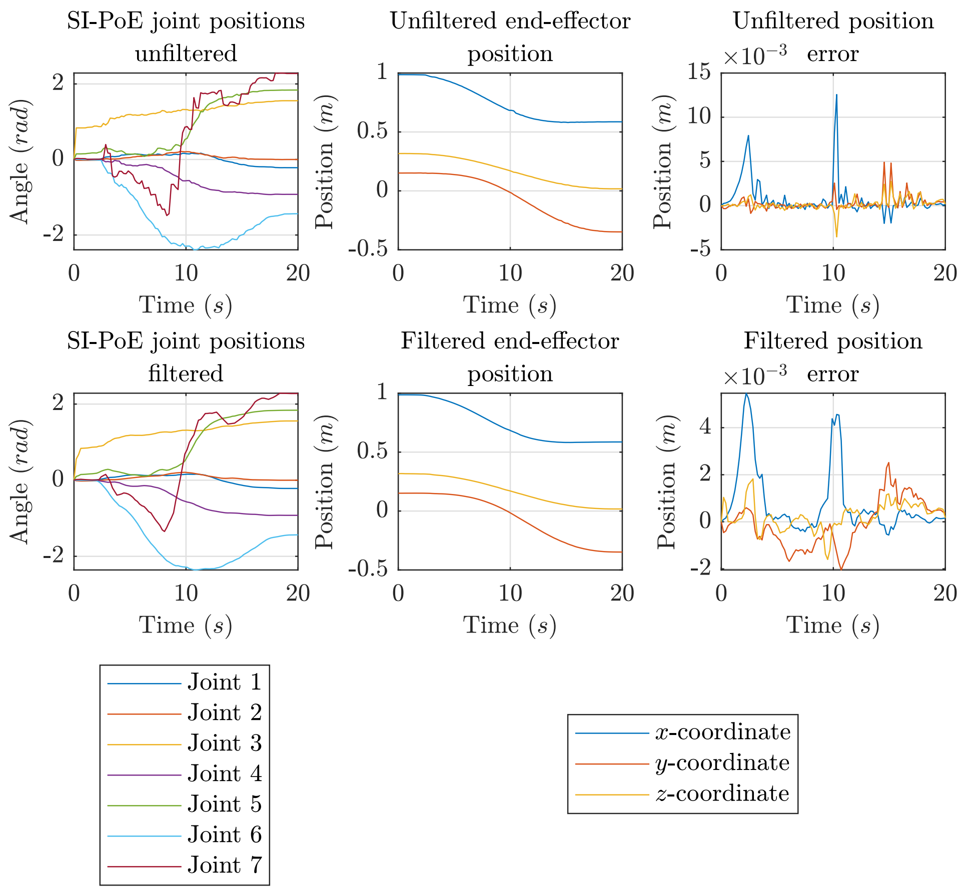

Figure 5.7 SI-PoE unfiltered and filtered joint positions and position errors.

The resulting joint trajectories obtained from the SI-PoE contained short term fluctuations, which were filtered using a moving average filter. The filtering takes negligible computational resources and requires approximately $0.02\ s$ of computational time for a trajectory with 100 points. The raw and filtered joint trajectories along with their position error for the desired trajectory shown in Figure 5.2 are demonstrated in Figure 5.7. Applying the moving average filter smooths the trajectories and eliminates spikes at the expense of slightly increasing the overall position error, which happens due to "flattening" of the spikes. Along the trajectory, the maximum error reduces from approximately $13\ mm$ (unfiltered) to $4\ \ mm$ (filtered) at $10\ \ s$. The total error integrated over time in all 3 axes is $0.0363\ \ ms$ for the unfiltered joint trajectory and $0.0440\ \ ms$ for the filtered SI-PoE joint trajectory. The resulting movement of the simulated robotic arm is shown Figure 5.8, where it can be seen that the SI-PoE was able to successfully generate an obstacle avoidance trajectory with additional constraints such as the finite jerk on the end-effector and prioritized penalty on excessive joint movement. At approximately $10\ s$ the robotic arm's virtual points approach the third obstacle. In response, the robotic arm starts actively utilizing rolling joint number 4 in order to stop the movement of the robotic arm towards the obstacle and continue advancing the end-effector along the desired trajectory.

The finite jerk constraint can be applied on the joint trajectories after obtaining the trajectories from the SI-PoE, or separately if initial and final joint angles are known. Finite jerk joint angles, velocities, and accelerations as a function of time can be computed as follows using the Quintic Polynomial Finite Jerk (QPFJ) model:

$$\underline{\theta}_{\mathrm{fj}}(t) = \underline{\theta}_{\mathrm{end}}[(10/t_{\mathrm{end}}^{3})t^{3} - (15/t_{\mathrm{end}}^{4})t^{4} + (6/t_{\mathrm{end}}^{5})t^{5}] \tag{5.14}$$

$$\dot{\underline{\theta}}_{\mathrm{fj}}(t) = \underline{\theta}_{\mathrm{end}}[(30/t_{\mathrm{end}}^{3})t^{2} - (60/t_{\mathrm{end}}^{4})t^{3} + (30/t_{\mathrm{end}}^{5})t^{4}] \tag{5.15}$$

$$\ddot{\underline{\theta}}_{\mathrm{fj}}(t) = \underline{\theta}_{\mathrm{end}}[(60/t_{\mathrm{end}}^{3})t - (180/t_{\mathrm{end}}^{4})t^{2} + (120/t_{\mathrm{end}}^{5})t^{3}] \tag{5.16}$$

where $\underline{\theta}_{\text{fj}}$, $\dot{\underline{\theta}}_{\text{fj}}$, $\ddot{\underline{\theta}}_{\text{fj}}$ are finite jerk joint angles, velocities, and accelerations respectively; $\underline{\theta}_{\text{end}}$ is the final joint position obtained using SI-PoE; t_{end} is the total movement time.

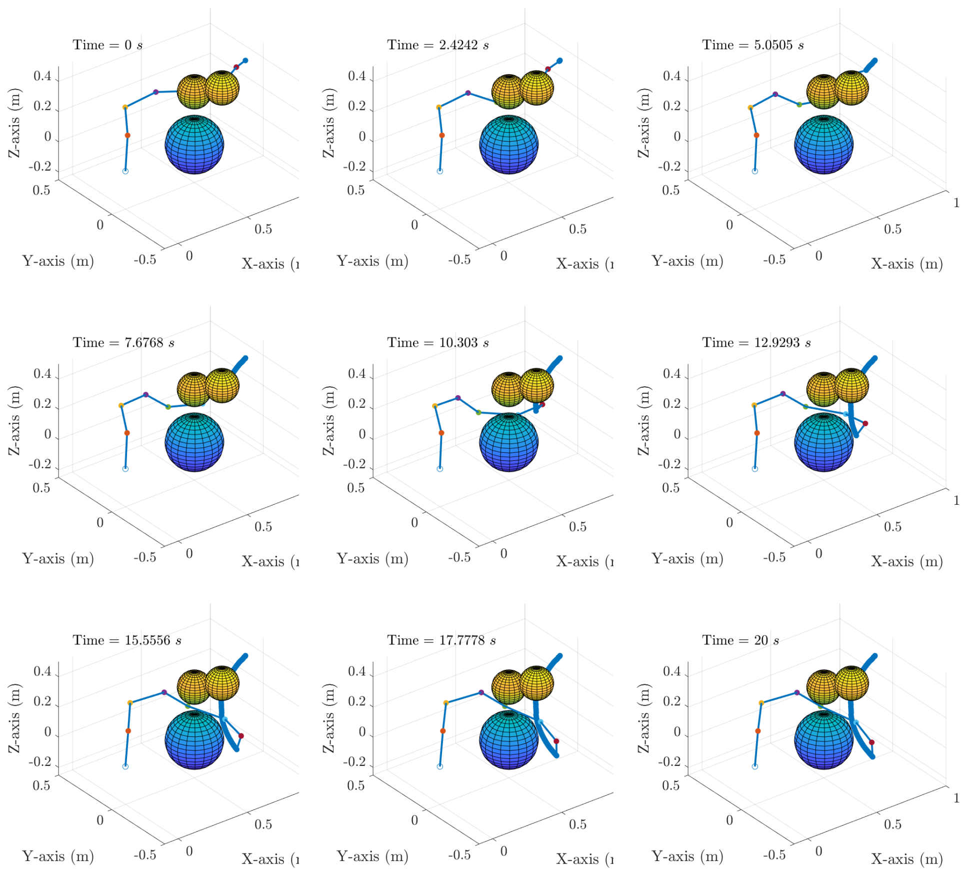

Figure 5.8 Obstacle avoidance trajectory generated using SI-PoE.

Although Equations 5.14 - 5.16 produce a joint position profile with finite jerk, the position error to the desired path becomes very high as demonstrated in Figure 5.9, which would certainly lead to collision with obstacles in the workspace. One way of integrating the QPFJ model with the SI-PoE is to use several via-points on the SI-PoE joint position trajectory and “sew” the trajectory from the finite-jerk profile piece-by-piece. The downside of this “sewing” approach is that it requires additional computational effort.

5.4. Implementation

Experiments were conducted on the 7-DoF Sawyer robotic arm, shown in Figure 5.10. For implementation and validation purposes, the joint positions obtained using SI-PoE were directly fed to the Sawyer robotic arm through the Robot Operating System (ROS) - Python environment on the Software Development Kit (SDK) mode. SI-PoE and QPFJ trajectores shown in Figure 5.9 were tested. Joint angles, velocities, and torques were recorded. The end-effector position error was calculated from the experimental joint angles.

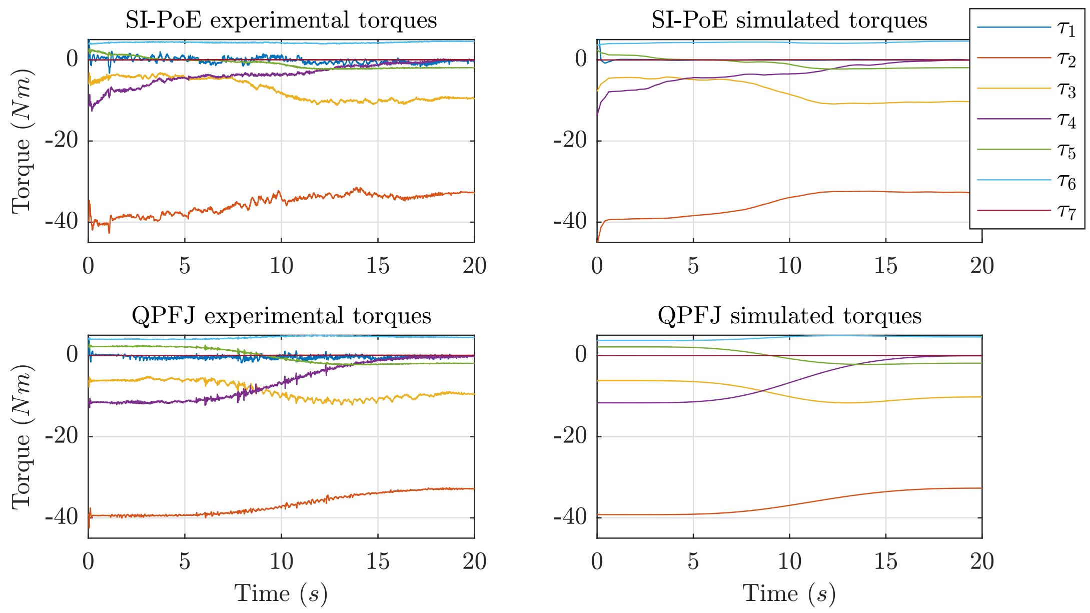

Figure 5.11 Experimental and simulated QPFJ and SI-PoE torques.

The experimental results for the joint angles and end-effector position are shown in Figure 5.12. The Sawyer robotic arm successfully tracked both the SI-PoE and QPFJ trajectories with minimum noise introduced. This result is evident in Figure 5.12, where the maximum position error of SI-PoE in any axis at any point of time within the trajectory is approximately $6\ mm$ which correlates well with the simulated trajectory shown

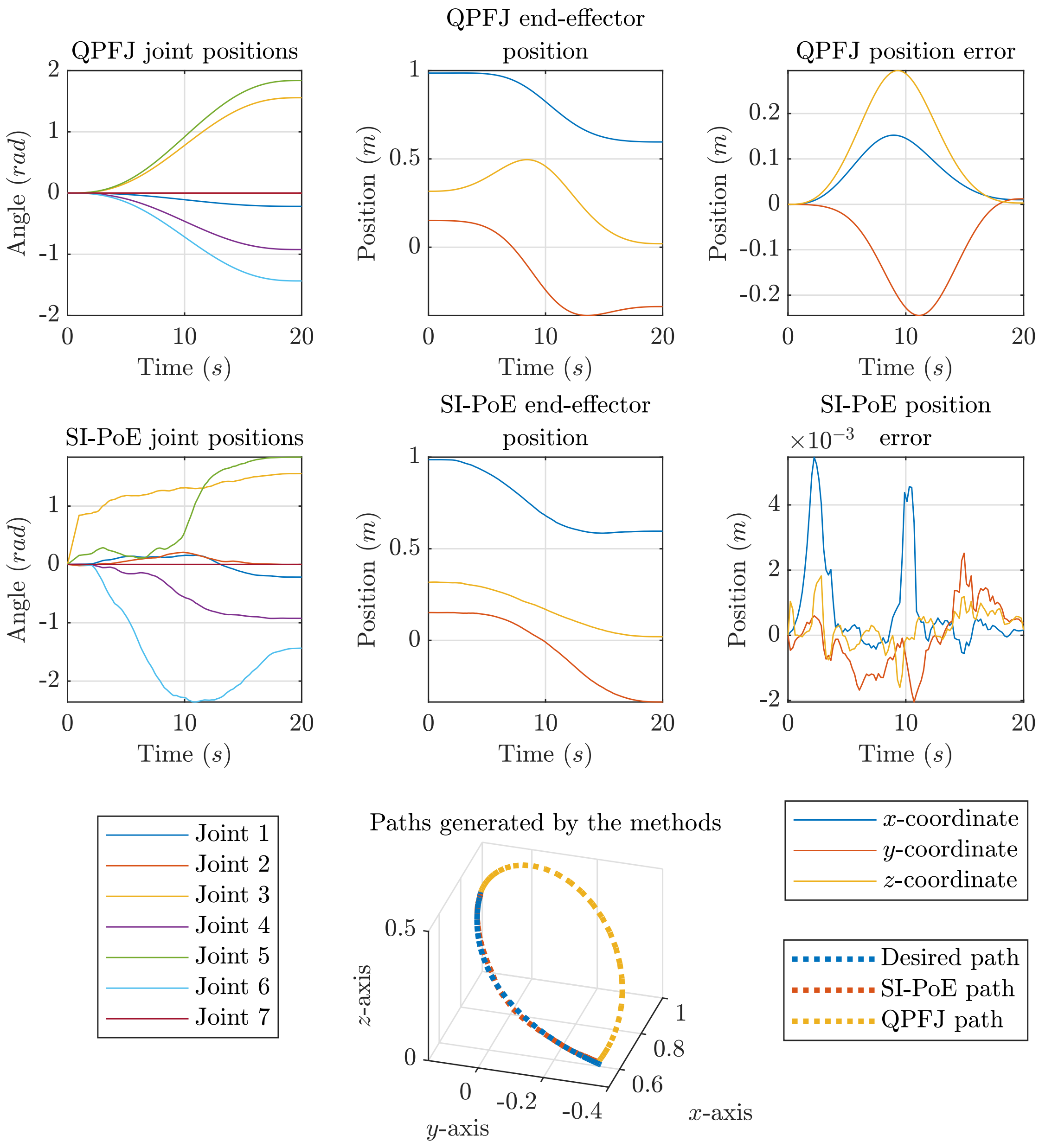

Figure 5.9 Simulated QPFJ and SI-PoE trajectories compared with the desired path.

in Figure 5.9. The integrated error in all axes is $0.1172\ \ ms$, which is larger than the simulated integrated error of $0.0440\ \ ms$, which is attributed to the noise in the robotic arm throughout the entirety of the trajectory. The Sawyer end-effector tracked the finite

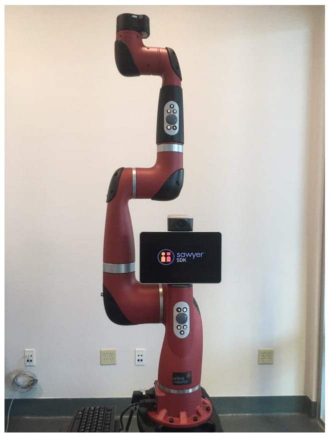

Figure 5.10 Sawyer robotic arm.

jerk profile well, while avoiding “virtual” obstacles. Eight virtual points were used for collision detection. The entire movement took 20 seconds, while the computation of the trajectory took approximately 13 seconds, indicating that the SI-PoE algorithm can be applied “online” while the robotic arm is operating, given that the number of virtual points is optimized a priori. Depending on the size of the obstacles in the workspace, the number of virtual points checked for collision should be changed or the surrounding boundaries of the obstacles should be increased in size. However, such methods could lead to a limited S-space which could result in undesired trajectories, errors in the position of the end-effector, or in critical cases even an absence of the solution.

The experimental torque profiles were compared with the simulated torque profiles for validation purposes. The inverse dynamics algorithm outlined in Chapter 3 was used to generate simulated joint torque profiles. Experimental torque profiles were also contaminated with noise. Nevertheless, the predicted joint torque trajectories approximate the experimental ones very well as seen in Figure 5.11. It should be noted that less noise was present in the QPFJ trajectories, which shows that, generally, smooth joint trajectories produce smooth torque trajectories. Torque trajectories with and without penalty on excessive joint movement were compared. Although the proposed SI-PoE succeeded in minimizing torque by minimizing the excessive joint movement, as was proposed in Equation 5.7, the difference was marginal with torque minimized roughly by $2 - 3\ Nm$ in most of the joints.

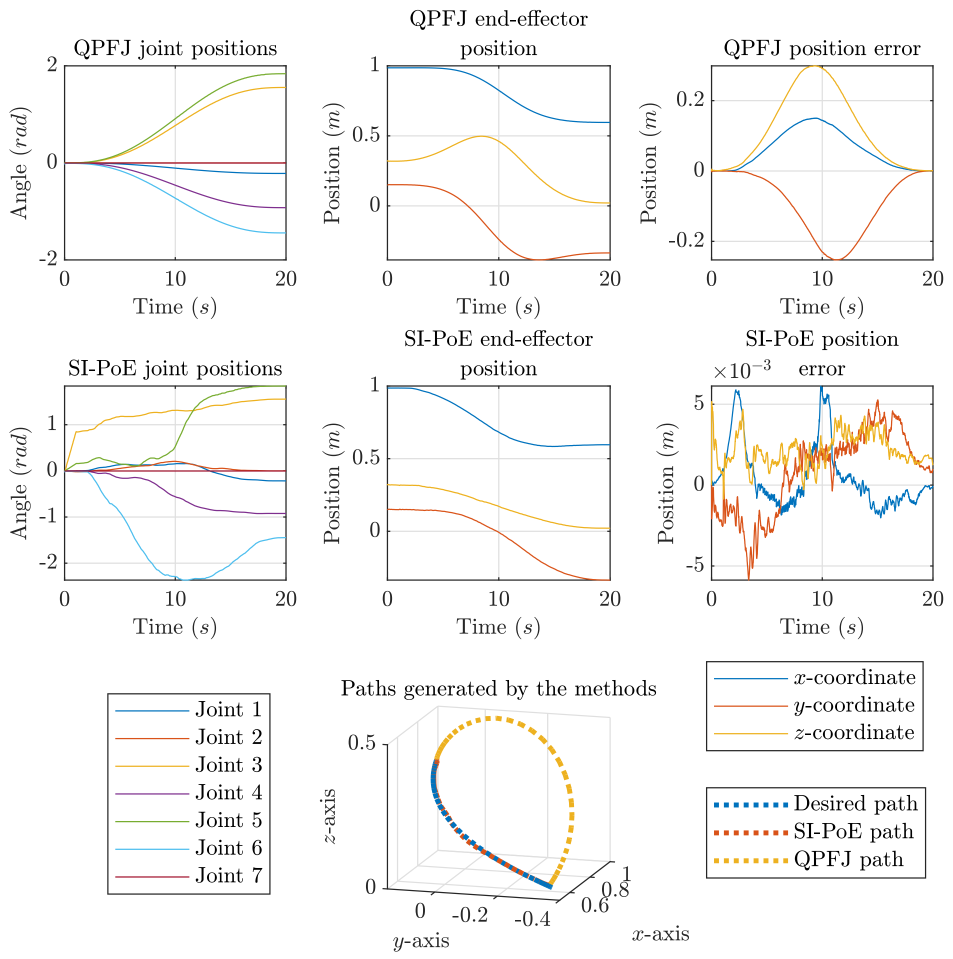

Figure 5.12 Experimental and simulated QPFJ and SI-PoE trajectories.

5.5. Discussion

In this chapter, an SI-PoE algorithm was outlined, simulated, and experimentally validated using the Sawyer robotic manipulator. The proposed SI-PoE algorithm utilizes

PoE-FK for forward kinematics computation and collision detection. The proposed fitness function of the SI-PoE includes both orientation and position errors, penalty for excessive joint movement, and collision avoidance. The algorithm's hyperparameters were demonstrated along with the proposed swarm initialization that improved both the computation time and accuracy of the end-effector position for each step. SI-PoE was able to satisfy multiple constraints such as avoiding obstacles in the workspace, minimizing excessive joint movement subsequently minimizing torque, and tracking a finite jerk end-effector trajectory, as demonstrated in the simulation and experimental validation with a maximum position error of $6\ mm$ at any time throughout the trajectory. The experimental torque profiles were compared with simulated inverse dynamics torque trajectories and showed good correspondence. SI-PoE can be used "online" if the number of virtual points checked for collision is carefully selected, as was demonstrated in this work. The disadvantage of this method of obstacle avoidance is that if obstacles in the workspace are small in size, it would either require more virtual points for the SI-PoE to track using PoE-FK, which would considerably increase the computation time, or increase the boundary surrounding the obstacle, increasing the effective size of the obstacles and decreasing the swarm's search space.

The finite-jerk in joint space was achieved using QPFJ method, which produced smooth joint and torque trajectories. Although the finite-jerk constraint was satisfied using QPFJ, the simultaneous obstacle avoidance was not achieved. The end-effector position error can be reduced by utilizing Constant Screw Axis (CSA) trajectories. Potentially, QPFJ can be combined with SI-PoE so that the manipulator avoids obstacles and satisfies finite-jerk on its joints. However, it is expected that such a combination would increase the computational effort. An optimal combination of SI-PoE and QPFJ can be explored in future work.

6. Deep Q-Learning Trajectory Generation

This Chapter outlines a Deep Reinforcement Learning (RL) approach for solving the Inverse Kinematics (IK) problem of a 7-Degree of Freedom (DoF) robotic manipulator using Product of Exponentials (PoE) as a Forward Kinematics (FK) computation tool and the Deep Q-Network (DQN) as an IK solver. The algorithm is designed to produce joint space trajectories from a given end-effector trajectory. Different network architectures were explored and the output of the DQN was implemented on the Sawyer robotic arm. The DQN was able to find different trajectories corresponding to a single Cartesian path of the end-effector. The network agent was able to learn random Bézier end-effector trajectories in a reasonable time frame with good accuracy, demonstrating that even though DQN is mainly used in discrete solution spaces, it could be applied for generating joint space trajectories.

6.1. Methodology

The 7-DoF Sawyer robotic manipulator by Rethink Robotics was employed for simulations and experiments. This arm is shown in Figure 1.1. The Sawyer robotic arm has seven links and seven independently actuated joints. The information regarding the arm's dimensions, weight, inertia properties, joint positions and limits, etc. was extracted from the Universal Robot Description Format (URDF) file (Robotics, 2017), and was used in the simulation, where it also served as an input for the RL model.

The PoE method was employed for the FK modeling, as it is easier to set up and does not limit a user to follow a strict convention, which is present when working with the more conventional Denavit-Hartenberg (D-H) approach to FK (Lynch & Park, 2017). In order to accurately track the position and orientation of any point on a robotic manipulator, such as the end-effector, the PoE-FK model only requires its home configuration and Screw axes represented in inertial (space) frame. The end-effector home configuration is represented as a 4×4 $SE(3)$ matrix and is shown in Equation 5.5. The detailed description of the process is outlined in Chapter 1. The Screw axes, given in Equation 1.13, are represented in the inertial frame as well. The PoE-FK of the end-effector is thus given as:

$$T_{s7} = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \cdots e^{[\mathcal{S}_7]\theta_3} M_{s7} = \begin{bmatrix} R_{s7} & \underline{p}_{s7} \\ 0_{1\times 3} & 1 \end{bmatrix} \in SE(3) \tag{6.1}$$

where $(\theta_1, \theta_2, \cdots, \theta_7)$ are the joint positions of the manipulator, and $R_{s7} \in SO(3)$ and $\underline{p}_{s7} \in \mathbb{R}^3$ are the orientation and position of the end-effector respectively.

Similar to Equation 6.1, the PoE-FK representation of any point on a robotic arm can be described if its home configuration is available. The general formula for the PoE-FK of an i-th point attached to a j-th link can be expressed as follows:

$$T_{si} = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \cdots e^{[\mathcal{S}_j]\theta_j} M_{si} \tag{6.2}$$

6.2. DQN-IK Algorithm

DQN belongs to a family of RL algorithms that works by performing continuous iterations over the problem space, collecting knowledge and selecting the appropriate action. Here, it is used to solve the IK problem by observing the current state of the robot and deciding how the joints should be oriented to achieve the smoothest and the most accurate trajectory. DQN by itself can be described as a merger between Q-Learning and neural networks. It has been shown that it can perform quite challenging and complex tasks such as mastering Go (Silver et al., 2016) and several Atari games (Mnih et al., 2013). Being able to operate in environments with vast solution spaces and state-action pairing, it is of great interest for robotics applications (X. Shi et al., 2020).

Similarly to the Q-learning algorithm, the DQN agent operates on the observation, action selection, action execution, and reward obtainment scheme, as defined by the Markov Decision Process (MDP) (Hester et al., 2017). This process is essential for the agent's decision making and is described as the tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}\}$. Here, $\mathcal{S}$ denotes states on an agent in an environment, $\mathcal{A}$ denotes available actions, $\mathcal{R}$ represents rewards given to an agent, and $\mathcal{T}$ is the transition from one state to another. As for Q-learning, the goal of an agent is to observe the current state and select actions that would maximize the reward. To identify those actions, Q-learning uses a Q-table that maps states to actions and expected rewards. These expected rewards in the Q-table are specific for each state and are known as Q-values, being calculated using the Bellman equation:

$$Q(s, a) = Q(s, a) + \alpha(R + \gamma \max Q(s', a') - Q(s, a)) \tag{6.3}$$

where $Q(s, a)$ is the Q-value with the current state s and action a, α is the learning rate of an agent, R is the reward, γ is the discount factor, and $Q(s', a')$ is the reward estimation for the next state. When the agent is placed in an environment, it observes the environment via examining the Q-table, chooses actions, performs them, calculates the corresponding Q-value and updates the Q-table accordingly. It iterates over that process until it reaches the final state, i.e. its end goal. This algorithm is simple to implement and is generally efficient for environments with limited number of states and actions. However, it is not entirely suitable for complex environments, where the number of states is large, or, conversely, small but continuous (X. Shi et al., 2020). To realize the benefits of Q-learning in more complex environments, a reinforcement algorithm, such as DQN, can be used.

As mentioned earlier, DQN operates in a similar manner to Q-learning; however, the Q-table is replaced with a neural network, with a selected number of nodes and hidden layers, where weights and biases of these nodes represent Q-values. An additional benefit of DQN lies in a replay buffer, which allows to save some experience and reuse it during training (Hester et al., 2017). This allows to decrease training time and increases stability of an agent. The Bellman equation is adapted for neural networks and represents the squared loss function, as seen in Equation 6.4:

$$\mathcal{L}(s, a|\theta) = (R + \gamma \max Q(s', a'|\theta') - Q(s, a|\theta))^2 \tag{6.4}$$

where θ denotes current weights and biases of the network, R is the reward, and γ is, again, the discount factor. Note that $Q(s', a'|\theta')$ represents these parameters, such as states, actions, weights, and biases of the target network, and $Q(s, a|\theta)$ are the predicted values. In its operation in the environment, DQN performs gradient descent over the neural network, calculating the loss and choosing actions to minimize it, maximizing the reward.

6.2.1. Fitness Function for DQN-IK

To implement the DQN agent for solving the IK problem for the 7-DoF robotic manipulator, a simulation was performed using MATLAB with the Deep Learning Toolbox. The benefit of the simulation lies in its quick visualization of the solution as well as inclusion of all required dependencies. The simulation would provide the trajectory estimates that would then be used for visualization and could be fed to the real 7-DoF robotic arm. To accomplish that, a custom environment was created for the agent, where all initial properties were defined for the end-effector frame and joint space position parameters θ. A total of seven θ parameters were present, one for each joint; these will serve as states for the DQN agent. This $\underline{\theta}$ vector was required to be found by an agent for every point in the trajectory via performing the gradient descent over the neural network. After finding and tuning optimal position parameters for one point, the agent would move to the next, compiling all points at the end into the full trajectory matrix. Actions were defined as incremental changes to θ by a certain amount $\delta\theta = 0.0025°$. The training time is very sensitive to the $\delta\theta$ parameter.

In general, a total of three types of actions were present: increase θ, decrease θ, or leave it unchanged. As each of these action types can be applied to each joint, the number of all possible combinations equals $3^7 = 2187$. Upon executing an action by updating joint space position parameters, the orientation was calculated using the PoE-FK model (Equation 6.1). The output of the latter was used to evaluate the fitness of the selected action, which would lead to the reward selection.

The fitness itself represents how close the position of the end-effector is to the desired point in the path. Several fitness functions were implemented, where only some checked the position error, and others evaluated the configuration error (position and attitude errors combined). All of the attempted fitness functions are shown in Equations 6.5 through 6.8, where the fitness value is a scalar for all cases (Malik, Lischuk, et al., 2021a).

$$f_i^j = ||\underline{p}_i^j - \underline{p}_d^j|| \tag{6.5}$$

$$f_i^j = \mid [1\ 1\ 1]\underline{p}_i^j - [1\ 1\ 1]\underline{p}_d^j \mid \tag{6.6}$$

where $\underline{p}_d^j$ is the (d) desired j-th point's Cartesian coordinates along the trajectory, and the $\underline{p}_i^j$ is the algorithm's i-th iteration of the corresponding point's coordinates. The fitness functions shown in Equations 6.5 and 6.6 are rather simplistic, where only the norm mapping and absolute value operators were used. These functions lead to non-monotonic convergence where the fitness value would oscillate about a certain value close to the goal fitness f_g, and most of the time diverging as the number of iterations increase. Also, these fitness functions only take into account the position and do not attempt to minimize the attitude error. The following fitness functions were also implemented:

$$f_i^j = \| \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \log(T_{s7}^{-1}(\theta_i^j)T_{sj})^{-\wedge} \| \tag{6.7}$$

$$f_i^j = \| \log(T_{s7}^{-1}(\theta_i^j)T_{sj})^{-\wedge} \| \tag{6.8}$$

where T_{sj} is the configuration of the desired j-th point along the trajectory, $T_{s7}^{-1}(\theta_i^j)$ is the algorithm's corresponding i-th iteration, $\log$ is the matrix logarithm, and $(-\wedge)$ symbol represents the "un-wedge" mapping from $se(3)$ to $\mathbb{R}^6$. The fitness functions represented in Equations 6.7 and 6.8 result in monotonic convergence to the fitness goal value f_g with occasional bounces just around the desired fitness value. Also, the fitness function given in Equation 6.8 takes into account both the position and attitude error, such that the algorithm can be configured to learn to track both the Cartesian trajectory of an end-effector and the attitude.

6.2.2. Reward, Network Architecture, and Hyperparameters

After calculating the fitness, a reward was given to the agent. The reward itself was adjusted dynamically according to the value of the aforementioned fitness. For example, if the fitness of the current iteration after adjusting positions is lower than the fitness for the previous iteration, then reward is positive, negative if otherwise, and close to no reward if it is exactly the same. To ensure the reward is given in correspondence to the difference between the previous and current fitness, it has been modeled using the $\arctan$ function in a range of $(-\pi/2, \pi/2)$ (Hu et al., 2019). This approach avoids possible overrewarding or overpunishing the agent. Therefore, to incorporate the difference between previous and current fitness, as well as the desired goal, the reward is defined as:

$$R = \arctan\left((f_{i-1}^{j} - f_{i}^{j})\frac{\pi}{2}\frac{1}{f_g}\right)m, \tag{6.9}$$

where f_{i-1}^{j} is the fitness for the previous iteration for the point j, f_{i}^{j} is the fitness for the current iteration for the point j, f_g is the goal fitness, and m is the selected magnification factor.

The goal fitness represents the fitness value that was manually chosen after some experimentation; it is used to stop calculations for the current point and move to the next, which affects both the accuracy of tracking and training time. This value was chosen to be ≈ 0.005. The magnification factor m is another chosen parameter used to increase the output of the $\arctan$ as fitness values tend to be on the order of 10^{-3}. Based on experimentation, the factor was selected as $m = 10$. After obtaining the reward, the DQN agent will store its experience in the so-called Experience Buffer. This buffer is also used by the agent to sample small batches of the $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}\}$ tuple to essentially replay some experience, allowing to learn more from the same data. Lastly, the agent will calculate the overall network loss using Equation 6.4 and update weights and biases of the network accordingly. The general algorithm of how the DQN agent operates in an environment for finding the trajectory is presented in Algorithm 1.

The model architecture consists of four main layers: an input layer, two hidden layers, and an output layer. As was mentioned earlier, seven joint space positions θ served as

Algorithm 1 Pseudo-code for finding joint positions on the trajectory using DQN.

```
Initialize custom environment for the DQN agent.
Set the initial point of the screw axis that must be the closest to the Bézier curve.
Initialize the DQN Critic network.
Reset initial states: joint positions $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}$ for TrajectoryPoint $j = 1$.

for TrajectoryPoint $j = 2$ to MaxTrajectoryPoints do
    Observe states $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}$ from TrajectoryPoint $j - 1$.

    for Episode $i = 1$ to MaxEpisodes do
        Observe states $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}$.
        Select action based on DQN Critic network's weights and the exploration factor.
        Update states $\{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, \theta_7\}$ by amount $\delta\theta$ according to the selected
action.
        Perform action via:
            $$T_{si} = e^{[\mathcal{S}_1]\theta_1} e^{[\mathcal{S}_2]\theta_2} \cdots e^{[\mathcal{S}_j]\theta_j} M_{si}$$
        Evaluate fitness:
            $$f_i^j = \| \log(T_{s7}^{-1}(\theta_i^j) T_{sj})^{-\wedge} \|$$
        Obtain reward:
            $$R = \arctan\left( (f_{i-1}^j - f_i^j) \frac{\pi}{2} \frac{1}{f_g} \right) m$$
        Store the experience in ExperienceBuffer.
        Sample random minibatch of transitions from the ExperienceBuffer.
        Calculate DQN Critic network loss $\mathcal{L}$ and update network weights.
        if Fitness == Goal then
            $break$
        end if
    end for
end for
```

states, and, architecturally speaking, inputs to the neural network. It is followed by a first fully connected, or, in other words, a dense layer with 128 nodes accompanied by a Rectified Linear Unit (ReLU) activation function. This function was chosen due to its computational efficiency and general popularity. Next, another fully connected layer was included with 256 nodes and another ReLU function. Finally, the output layer represents actions with a total of 2187 nodes, one for each action type available for each joint. This architecture can be seen in Figure 6.1. Table 6.1 provides a summary

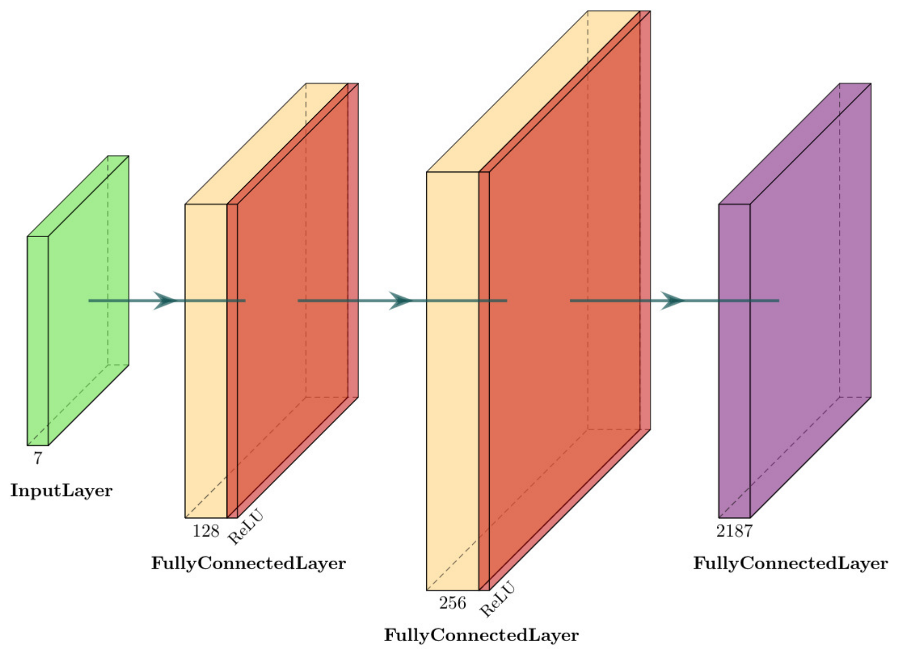

Figure 6.1 DQN model used for solving inverse kinematics.

of other parameters, such as the learning rate, discount factor, gradient threshold, as well as selected parameters for the reward calculation per Equation 6.9.

Table 6.1

Summary of selected parameters for the DQN agent and reward function

Parameter	Value
Joint Space Position Step $\delta\theta$	$\pm 0.0025°$
Learning Rate	0.01
Discount Factor	0.9
Gradient Threshold	1
Mini Batch Size	64
Target Smooth Factor	1×10^{-3}
Goal Fitness f_g	0.005
Magnification Factor m	10

6.3. Implementation

A workstation with an Intel® Core™ i5-10600K Processor, 16 GBs of RAM was used to host the simulation of the DQN algorithm for solving IK in MATLAB. For the sake of brevity, this DQN agent along with IK computation and solution can be referred to as DQN-IK. Random Bézier curves were generated and provided as an input to the simulation, and the average time for the algorithm to learn a trajectory consisting of 100 Cartesian points is about 30-40 minutes on this particular workstation. The DQN-IK is a bit slow at the beginning of the trajectory for the first 3-4 points and speeds up for the rest, which owes to the aforementioned Experience Buffer. The hyper-parameters, presented in Table 6.1, were obtained by rigorous studies and performance comparison. The goal fitness value of $f_g = 0.005$ led to a maximum allowable error in end-effector position of $5\ mm$. Despite that, this fitness can be increased to speed up the learning of the DQN agent, but that would lead to inaccurate positioning of the end-effector. Conversely, the smaller the goal fitness, the more accurate the positional tracking is and the longer it takes for the agent to learn.

The architecture of the neural network (Figure 6.1) was also tuned to strike a balance between the complexity and the speed of the algorithm. As was mentioned in Section 6.2.2., the number of hidden layers was chosen to be two with 128 and 256 nodes respectively. This number of nodes was determined from experimentation and found to provide a good balance between accuracy of the solution and computation speed. Increasing the number of nodes as well as layers did not provide sufficient gains. A lower number of nodes, 64 and 128 can be used as well, but at a minor cost in accuracy.

Regarding the number of hidden layers, one layer will not be sufficient, as the DQN agent will start having trouble finding the solution; thus having two layers provides a reasonable implementation.

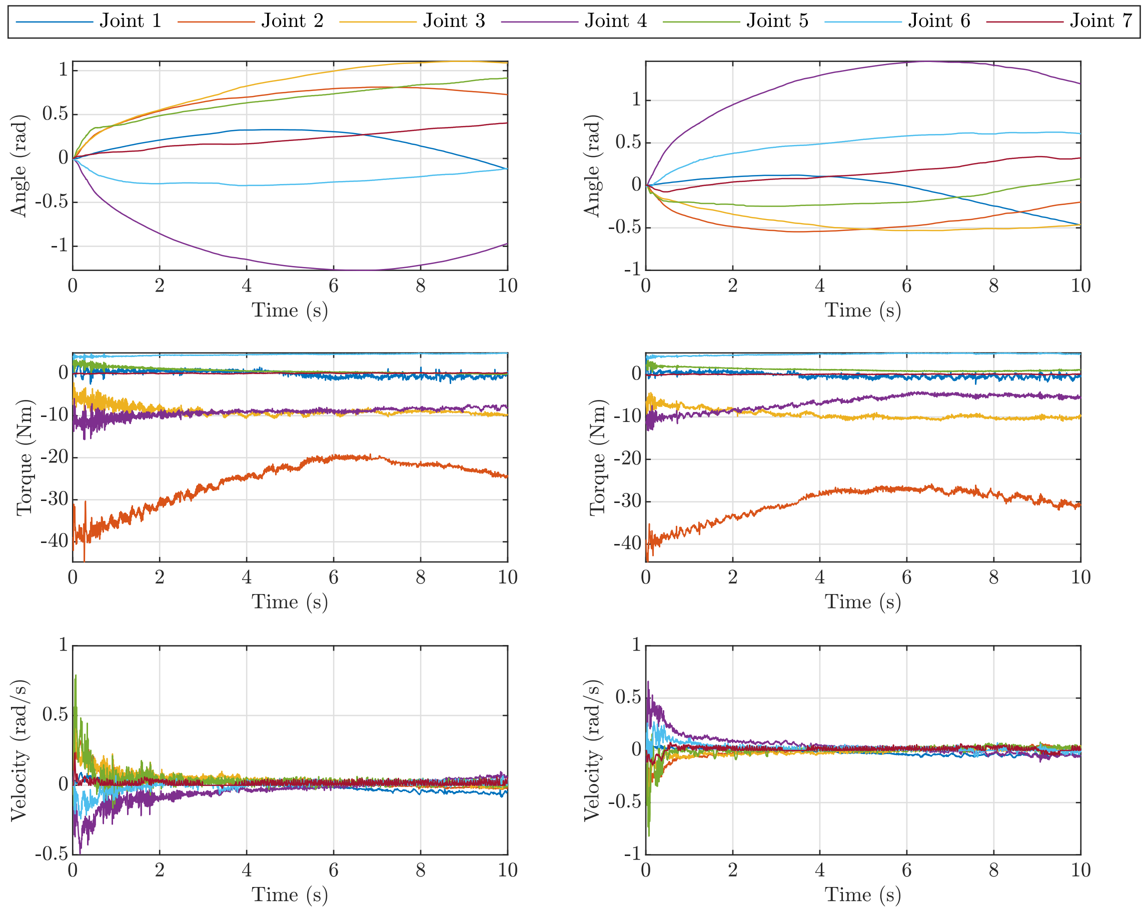

Figure 6.2 Experimental trajectories recorded from the Sawyer robotic arm.

As there are infinite solutions in the joint space, the DQN-IK algorithm was able to generate several different joint space trajectories corresponding to a single Bézier curve path. This is demonstrated in Figure 6.2, where it is evident that the algorithm approaches the same path with different solutions. The heatmaps were different for the same path as well. A sample heatmap is shown in Figure 6.3, and a corresponding bar chart is shown in Figure 6.4, where it can be seen that, in this particular case, the algorithm has utilized a couple distinct actions more than the others.

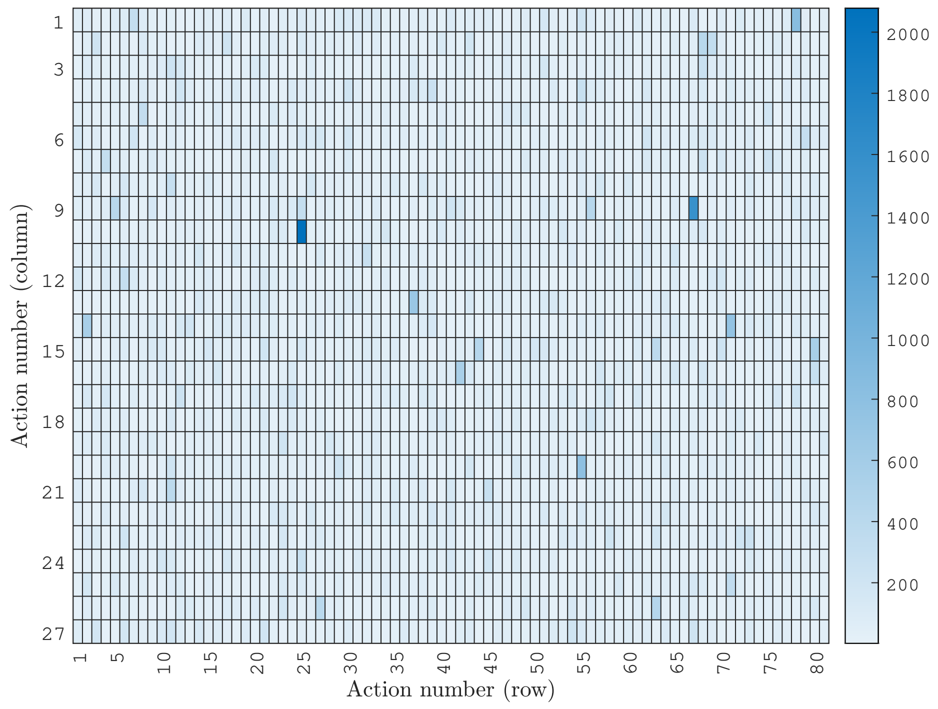

Figure 6.3 Heatmap corresponding to a DQN-IK trajectory.

The experimental implementation was carried out by sending the joint trajectory data to the Sawyer robotic arm through the Ubuntu-ROS-Sawyer framework, where the commands were provided as an input for the Python-based controller and sent to the robotic arm in the Software Development Kit (SDK) mode. Joint angle positions, velocities, and efforts (torques) were recorded and are shown in Figure 6.2. The recorded experimental joint space trajectories are almost indistinguishable from the DQN-generated profiles. The resulting end-effector Cartesian path is demonstrated in Figure 6.5. As shown in this figure, the algorithm has found different solutions: in the first one the manipulator tracks the trajectory from “below” (left column) and in the second from

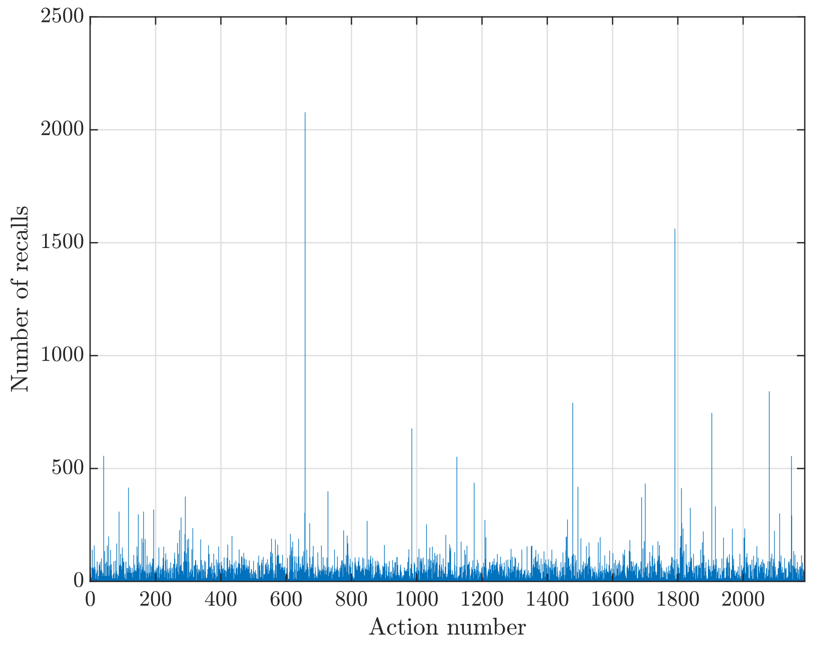

Figure 6.4 Bar chart of the DQN-IK trajectory.

“above” (right column). In Figure 6.6 it is demonstrated that the Cartesian error stays within the bounds of $\pm 5\ mm$ except the initial jolt where the error spikes up to $10\ mm$ for a fraction of a second. The DQN-IK algorithm can also be utilized for generating trajectories to avoid obstacles in the task space. To perform this task, the fitness function should be modified to take into account the collision condition, which could be manifested by a large increment in the fitness. The collision condition can be tracked using the PoE-FK model, shown in Equation 6.1. Generally, the outlined algorithm is very versatile and modular as various constraints can be incorporated to meet different design objectives.

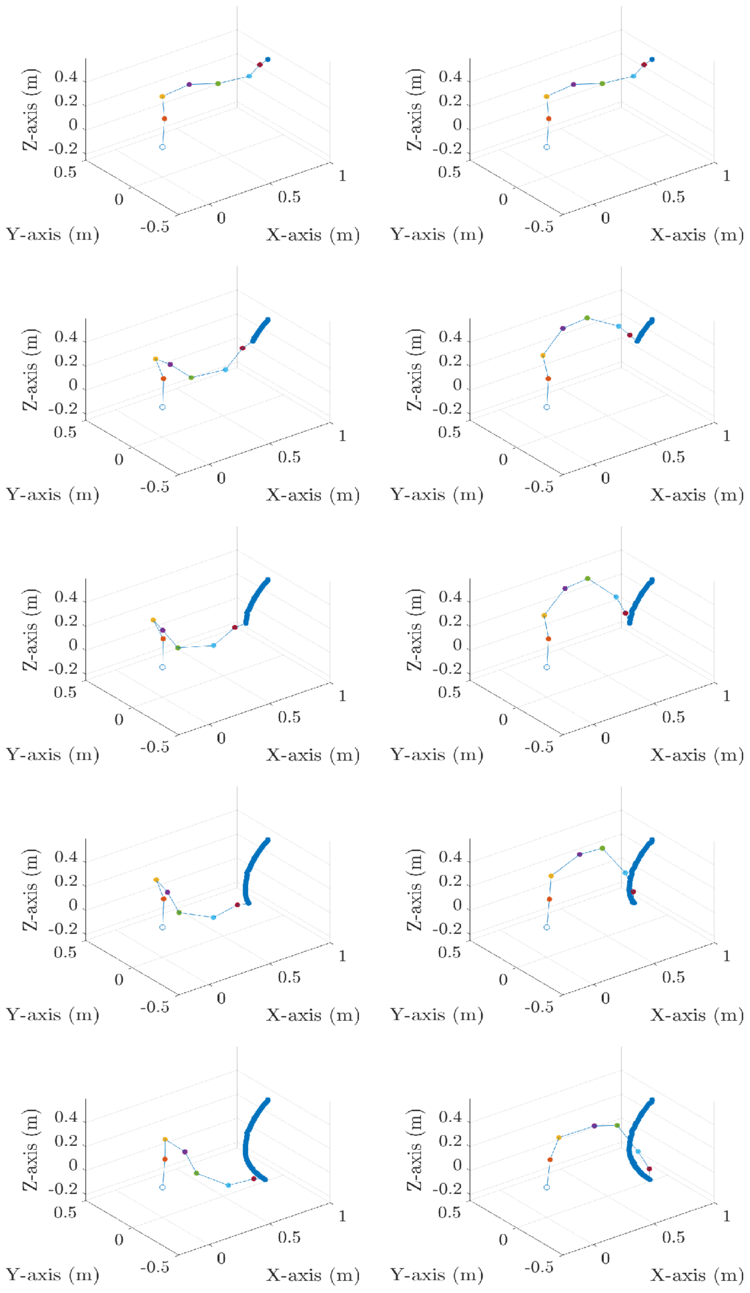

Figure 6.5 Sawyer robotic arm DQN-IK solution.

6.4. Discussion

In this chapter, it was shown that the Deep Reinforcement Learning approach using DQN is viable for solving the inverse kinematic problem of a high-DoF robotic manipulator. In general, the tracking position and/or orientation of any point or joint of the robotic manipulator only requires the home configuration of the point and Screw axes, which are constant vectors. The proposed approach was used in conjunction with the DQN algorithm to calculate optimal joint positions to achieve the required trajectory.

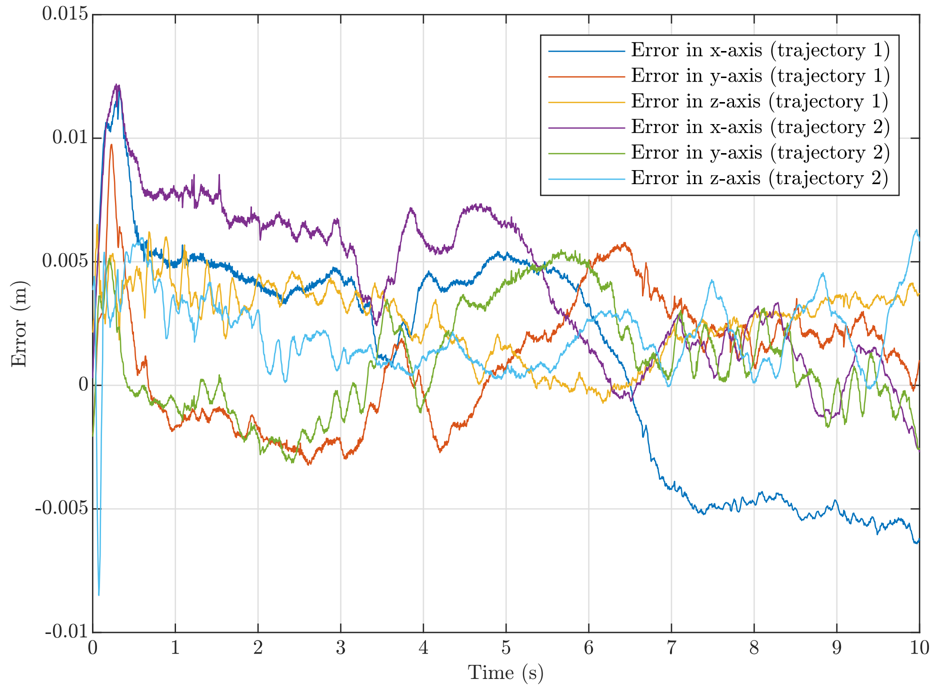

Figure 6.6 Error in Cartesian components of the end-effector.

The DQN-IK algorithm takes in joint space position parameters as inputs and selects an appropriate action to adjust them in order to track Cartesian points on the desired trajectory. The algorithm itself is comprised of the neural network with four layers: input, two hidden layers and one output layer. During operation and for each iteration, the DQN agent attempts to select appropriate action in terms of increasing/decreasing certain joint space position values by a specified amount. The algorithm is then evaluated on its action via calculation of the fitness value. The aim of this fitness is to calculate and demonstrate how close the posture described by all seven joint positions is to the desired point on the

Cartesian trajectory. This fitness is compared to the value of the previously calculated fitness and the reward is given accordingly. The reward is dynamically adjusted based on how far the previous fitness is to the newly calculated one. This allows to guide the DQN agent more closely to its goal, decreasing computation time without giving too much reward/punishment to the agent while trying to determine and optimize joint positions, and to increase the convergence rate.

The movement of the 7-DoF robotic arm was simulated using MATLAB and then, using generated data, was executed on the Sawyer robotic manipulator. During the simulation, it was observed that, due to the infinite number of solutions in the joint space, the DQN algorithm can find new trajectories each time it runs. It was demonstrated that even though DQN, by itself, is bound to discrete action spaces, using DQN for inverse kinematics proved to be a viable option, especially considering that its architecture is less complicated than DDPG or NAF. The more elegant approach may still lie in using DDPG as its action spaces are continuous. Nevertheless, while being more simple architecture-wise than DDPG, DQN proved to provide a solution with good accuracy in a reasonable computation time, and can be modified to fit different needs and objectives. The parameters selected for the neural network after extensive testing and optimization are presented in this paper as well, but can be adjusted to further optimize the algorithm and decrease computation time.

7. Conclusions and Future Work

Solving the Inverse Kinamtics (IK) problem is one of the biggest challenges that must be addressed to achieve full autonomy of complex robotic systems such as high DoF robotic manipulators. This work is a contribution towards the practical use of modern tools introduced in Chapter 2 for trajectory generation algorithms that solve the IK problem and produce joint space trajectories from a given Cartesian trajectory of an end-effector. All of the algorithms were based on the Product of Exponentials Forward Kinematics (PoE-FK) model. It was shown that PoE-FK is a versatile novel tool that can be used in combination with numerical, iterative, optimization, and intelligent methods. An iterative Newton-Raphson, Swarm Intelligence Product of Exponentials (SI-PoE), and reinforcement learning Deep Q-Learning Inverse Kinematics (DQN-IK) algorithms were developed and applied to a 7-DoF open-chain manipulator, resulting in desired behavior. This chapter summarizes what was achieved and suggests directions for future research.

7.1. Summary

Chapter 3 introduced the first algorithm based on the Newton-Raphson iterative method which solves the IK problem iteratively to generate constrained joint-space trajectories corresponding to straight-line and curvilinear motions of the end effector in Cartesian space with finite jerk. Derivatives of these joint-space trajectories are computed using Bézier curves, which ensures dynamically feasible trajectories. Since it is a numerical method the accuracy of the derivatives of joint-space trajectories was assessed using a novel method based on Mean Arctangent Absolute Percentage Error (MAAPE). Joint space trajectories produced by the Newton-Raphson IK algorithm were successfully implemented on the Sawyer robotic arm resulting in accurate tracking of the end-effector trajectory.

In Chapter 4, Constant Screw Axis (CSA) trajectories were explored. The main challenge associated with CSA trajectories is that, for a redundant system like the Sawyer robotic arm, there are infinite trajectories that follow the CSA profile. Finding the most suitable one was achieved by applying several different modern tools. Particle Swarm Optimization (PSO), Q-Learning, and Deep Q-Learning (DQN) were used for that purpose, and all of these tools were able to successfully obtain a CSA trajectory that was closest to the desired Cartesian trajectory of the end-effector. In this particular case, PSO proved to be the fastest and the most reliable tool.

Chapter 5 described a multi-objective IK solving algorithm which simultaneously accurately tracks the end-effector trajectory, satisfies the finite jerk constraint, minimizes joint movement and joint effort, and avoids obstacles. All of these objectives were achieved by combining the Swarm Intelligence with Product of Exponentials (SI-PoE). Different layouts of obstacles were checked in the simulations, which resulted in different joint space trajectories, all of which successfully avoided collision in the work space. The main setback of this algorithm is that the computational cost of the solution increases as more accurate obstacle avoidance is desired as the collision detection accuracy depends on the number of virtual points that are solved using PoE-FK. The output of the SI-PoE algorithm was tested on the Sawyer arm, resulting in accurate tracking of the end-effector Cartesian trajectory.

Robots can learn from experience through reinforcement learning techniques. Chapter 2 Sections 2.4.2. and 2.4.3. discussed several intelligent algorithms that can learn from off-policy actions in discrete and continuous state and action spaces. The ability to learn from experience replay, other behaviors, virtual targets, and other robotic systems are important for robotics as they can reduce experimentation time and lead to a fully autonomous systems. Robotics problems usually include continuous state and action variables. The solution space of high DoF robotic manipulators becomes very large making the discrete state and action methods very challenging to apply and tune. However, as the computational effort becomes less of an issue with modern hardware, discrete methods might become more attractive since their architecture is less complicated compared to the continuous methods. In Chapter 6, a Deep Q-Learning (DQN) method combined with PoE was used to solve the IK problem (DQN-IK). The DQN-IK agent was able to generate joint space trajectories for a 7-DoF Sawyer robotic arm in a reasonable amount of time with good accuracy. These trajectories were also experimentally tested on the Sawyer robotic arm that resulted in successful tracking of the end-effector trajectory.

7.2. Future Directions

The proposed trajectory generation algorithms can be improved further by incorporating additional tools to the developed PoE-FK model. Combining CSA or QPFJ trajectory properties with SI-PoE or DQN-IK could be a beneficial prospect if done correctly. In the optimal case, the latter algorithms would generate the “backbone” trajectories, which could be subsequently improved by the former tools by smoothing trajectory profiles leading to less vibration and torque efforts. The downside of this approach would be the added computational effort as the introduction of CSA/QPFJ trajectory properties will manifest itself as adding additional post-processing filtering. Nevertheless, such a combination might still be computationally less expensive than continuous actions and state RL agent.

The RL model for the IK solution can be greatly improved if the architecture is switched from the discrete actions and states DQN to continuous DDPG or NAF. Continuous state and action RL is worthwhile as smooth, accurate movement of the end-effector and the rest of the joints will become possible and practical. However, the policies learned from continuous models do not have convergence guarantees, and may introduce more architecture complexities. A more stable continuous action space algorithm could be developed if the RL model's policies assimilate ideas from conventional approaches to IK solving. As an example, the starting policy of a RL agent could be the minimization of a spatial twist.

The simulation and experimental implementations presented in this work were produced by learning from scratch. It results in slow learning, and sometimes even re-learning what is already known. Thus, introducing transfer learning or "initial guess" policies and NNs could potentially lead to shorter learning times and thus is suggested as a strong focus for future research.

REFERENCES

Ab Aziz, N. A., & Ibrahim, Z. (2012). Asynchronous particle swarm optimization for swarm robotics. *Procedia Engineering*, *41*, 951–957. doi: 10.1016/J.PROENG.2012.07.268

Abdi, A., Adhikari, D., & Park, J. H. (2021). A novel hybrid path planning method based on q-learning and neural network for robot arm. *Applied Sciences*, *11*(15), 6770. doi: 10.3390/APP11156770

Abdi, H. (1994). A neural network primer. *Journal of Biological Systems*, *2*(03), 247–281. doi: 10.1142/S0218339094000179

Atyabi, A., & Powers, D. M. (2013). Cooperative area extension of pso-transfer learning vs. uncertainty in a simulated swarm robotics. *International Conference on Informatics in Control, Automation and Robotics*, *2*, 177–184. July 29-31, Reykjavík, Iceland. doi: 10.5220/0004456901770184

Bellman, R. (1966). Dynamic programming. *Science*, *153*(3731), 34–37. doi: 10.2307/1909506

Bilbeisi, G., Al-Madi, N., & Awad, F. (2015). Pso-ag: A multi-robot path planning and obstacle avoidance algorithm. *2015 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, 1–6, 3-5 November, Amman, Jordan. doi: 10.1109/AEECT.2015.7360565

Bobrow, J. E., Dubowsky, S., & Gibson, J. S. (1985). Time-optimal control of robotic manipulators along specified paths. *The international journal of robotics research*, *4*(3), 3–17. doi: 10.1177/027836498500400301

Brockett, R. W. (1984). Robotic manipulators and the product of exponentials formula. *Mathematical theory of networks and systems*, 120–129. doi: 10.1007/BFB0031048

Casas, N. (2017). Deep deterministic policy gradient for urban traffic light control. *arXiv preprint arXiv:1703.09035*.

Chen, G., Wang, H., & Lin, Z. (2014). Determination of the identifiable parameters in robot calibration based on the poe formula. *IEEE Transactions on Robotics*, *30*(5), 1066–1077. doi: 10.1109/TRO.2014.2319560

Chen, R., & Dai, X.-y. (2018). Robotic grasp control policy with target pre-detection based on deep q-learning. *2018 3rd International Conference on Robotics and Automation Engineering (ICRAE)*, 29–33, 17-19 November, Guagzhou, China. doi: 10.1109/ICRAE.2018.8586758

Chiaverini, S., Oriolo, G., & Maciejewski, A. A. (2016). Redundant robots. In *Springer handbook of robotics* (pp. 221–242). Springer. doi: 10.1007/978-3-319-32552-1_10

Chirikjian, G. S. (2018). Discussion of "geometric algorithms for robot dynamics: A tutorial review"(fc park, b. kim, c. jang, and j. hong, 2018, asme appl. mech. rev., 70 (1), p. 010803). *Applied Mechanics Reviews*, *70*(1). doi: 10.1115/1.4039080

Collinsm, T. J., & Shen, W.-M. (2017). Particle swarm optimization for high-dof inverse kinematics. *2017 3rd international conference on control, automation and robotics (ICCAR)*, 1–6, 24-26 April, Nagoya, Japan. doi: 10.1109/ICCAR.2017.7942651

Couceiro, M. S., Machado, J. T., Rocha, R. P., & Ferreira, N. M. (2012). A fuzzified systematic adjustment of the robotic darwinian pso. *Robotics and Autonomous Systems*, *60*(12), 1625–1639. doi: 10.1016/j.robot.2012.09.021

Dadgar, M., Jafari, S., & Hamzeh, A. (2016). A pso-based multi-robot cooperation method for target searching in unknown environments. *Neurocomputing*, *177*, 62–74. doi: 10.1016/j.neucom.2015.11.007

Datta, B. N. (2010). *Numerical linear algebra and applications* (Vol. 116). Siam. doi: 10.1137/1.9780898717655

Deng, Y., Guo, X., Wei, Y., Lu, K., Fang, B., Guo, D., . . . Sun, F. (2019). Deep reinforcement learning for robotic pushing and picking in cluttered environment. *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 619–626. November 3-8, Macau, China. doi: 10.1109/IROS40897.2019.8967899

Dereli, S., & Köker, R. (2020). A meta-heuristic proposal for inverse kinematics solution of 7-dof serial robotic manipulator: quantum behaved particle swarm algorithm. *Artificial Intelligence Review*, *53*(2), 949–964. doi: 10.1007/s10462-019-09683-x

Eberhart, R., & Kennedy, J. (1995). Particle swarm optimization. *Proceedings of the IEEE international conference on neural networks*, *4*, 1942–1948. 27 November-1 December, Perth, Australia. doi: 10.1007/s11721-007-0002-0

Featherstone, R. (1983). The calculation of robot dynamics using articulated-body inertias. *The International Journal of Robotics Research*, *2*(1), 13–30. doi: 10.1177/027836498300200102

Featherstone, R. (2014). *Rigid body dynamics algorithms*. Springer. doi: 10.1007/978-1-4899-7560-7

Franceschetti, A., Tosello, E., Castaman, N., & Ghidoni, S. (2020). Robotic arm control and task training through deep reinforcement learning. *arXiv preprint arXiv:2005.02632*.

Gaskett, C. (2002). Q-learning for robot control. *The Australian National University*. (PhD dissertation) doi: 10.25911/5D7A2A09A7DFD

Gu, S., Holly, E., Lillicrap, T., & Levine, S. (2017). Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. *2017 IEEE international conference on robotics and automation (ICRA)*, 3389–3396. May 29 - June 3, Marina Bay Sands, Singapore. doi: 10.1109/ICRA.2017.7989385

Gu, S., Lillicrap, T., Sutskever, I., & Levine, S. (2016). Continuous deep q-learning with model-based acceleration. *International conference on machine learning*, 2829–2838. June 19 - 24, New-York, USA.

Guo, Z., Huang, J., Ren, W., & Wang, C. (2019). A reinforcement learning approach for inverse kinematics of arm robot. *Proceedings of the 2019 4th International Conference on Robotics, Control and Automation*, 95–99. July 26 - 28, Guangzhou, China. doi: 10.1145/3351180.3351199

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *International conference on machine learning*, 1861–1870. July 10-15, Stockholm, Sweden.

Hausknecht, M., & Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*.

Hays, C. W., Posada, D., Malik, A., Korczyk, D., Dafoe, B., & Henderson, T. (2021). Structural design and impact analysis of a 1.5u cubesat on the lunar surface. *31st AAS/AIAA Space Flight Mechanics Meeting*, February 1-3, Virtual.

He, C., Wang, S., Xing, Y., & Wang, X. (2013). Kinematics analysis of the coupled tendon-driven robot based on the product-of-exponentials formula. *Mechanism and Machine Theory*, *60*, 90–111. doi: 10.1016/J.MECHMACHTHEORY.2012.10.002

He, R., Zhao, Y., Yang, S., & Yang, S. (2010). Kinematic-parameter identification for serial-robot calibration based on poe formula. *IEEE Transactions on Robotics*, *26*(3), 411–423. doi: 10.1109/TRO.2010.2047529

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., … Dulac-Arnold, G. (2017). Deep q-learning from demonstrations. *arXiv preprint arXiv:1704.03732*.

Hoffman, J. D., & Frankel, S. (2018). *Numerical methods for engineers and scientists*. CRC press. doi: 10.1201/9781315274508

Hu, Z., Wan, K., Gao, X., & Zhai, Y. (2019). A dynamic adjusting reward function method for deep reinforcement learning with adjustable parameters. *Mathematical Problems in Engineering*, *2019*. doi: 10.1155/2019/7619483

Huang, H.-C., Xu, S. S.-D., & Hsu, H.-S. (2014). Hybrid taguchi dna swarm intelligence for optimal inverse kinematics redundancy resolution of six-dof humanoid robot arms. *Mathematical Problems in Engineering*, *2014*. doi: 10.1155/2014/358269

Hwang, K.-S., Tan, S.-W., & Chen, C.-C. (2004). Cooperative strategy based on adaptive q-learning for robot soccer systems. *IEEE Transactions on Fuzzy Systems*, *12*(4), 569–576. doi: 10.1109/TFUZZ.2004.832523

James, S., & Johns, E. (2016). 3d simulation for robot arm control with deep q-learning. *arXiv preprint arXiv:1609.03759*.

Joshi, S., Kumra, S., & Sahin, F. (2020). Robotic grasping using deep reinforcement learning. *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, 1461–1466. 20-21 August, Hong Kong, China. doi: 10.1109/CASE48305.2020.9216986

Khan, H., Abbasi, S. J., & Lee, M. C. (2020). Dpso and inverse jacobian-based real-time inverse kinematics with trajectory tracking using integral smc for teleoperation. *IEEE Access*, *8*, 159622–159638. doi: 10.1109/ACCESS.2020.3020318

Kim, S., & Kim, H. (2016). A new metric of absolute percentage error for intermittent demand forecasts. *International Journal of Forecasting*, *32*(3), 669–679. doi: 10.1016/J.IJFORECAST.2015.12.003

Klein, C. A., & Huang, C.-H. (1983). Review of pseudoinverse control for use with kinematically redundant manipulators. *IEEE Transactions on Systems, Man, and Cybernetics*(2), 245–250. doi: 10.1109/TSMC.1983.6313123

Konar, A., Chakraborty, I. G., Singh, S. J., Jain, L. C., & Nagar, A. K. (2013). A deterministic improved q-learning for path planning of a mobile robot. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *43*(5), 1141–1153. doi: 10.1109/TSMCA.2012.2227719

Korczyk, J. J., Posada, D., Malik, A., & Henderson, T. (2021). Modeling of an on-orbit maintenance robotic arm test-bed. *2021 AAS/AIAA Astrodynamics Specialist Conference*, August 8-12, Big Sky, USA.

Lee, H.-Y., & Liang, C.-G. (1988). A new vector theory for the analysis of spatial mechanisms. *Mechanism and Machine Theory*, *23*(3), 209–217. doi: 10.1016/0094-114X(88)90106-1

Lewis, M. A., & Bekey, G. A. (1992). The behavioral self-organization of nanorobots using local rules. *IROS*, 1333–1338. doi: 10.1109/IROS.1992.594558

Li, C., Wu, Y., Löwe, H., & Li, Z. (2016). Poe-based robot kinematic calibration using axis configuration space and the adjoint error model. *IEEE Transactions on Robotics*, *32*(5), 1264–1279. doi: 10.1109/TRO.2016.2593042

Li, W., & Xiong, R. (2019). Dynamical obstacle avoidance of task-constrained mobile manipulation using model predictive control. *IEEE Access*, *7*, 88301–88311. doi: 10.1109/ACCESS.2019.2925428

Li, Z., Ma, H., Ding, Y., Wang, C., & Jin, Y. (2020). Motion planning of six-dof arm robot based on improved ddpg algorithm. *2020 39th Chinese Control Conference (CCC)*, 3954–3959. 27-29 July, Shenyang, China. doi: 10.23919/CCC50068.2020.9188521

Liang, H., Lou, X., & Choi, C. (2019). Knowledge induced deep q-network for a slide-to-wall object grasping. *arXiv preprint arXiv:1910.03781*, 1–7.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, C.-J., Li, T.-H. S., Kuo, P.-H., & Wang, Y.-H. (2016). Integrated particle swarm optimization algorithm based obstacle avoidance control design for home service robot. *Computers & Electrical Engineering*, *56*, 748–762. doi: 10.1016/j.compeleceng.2015.05.019

Liu, Y.-C., & Huang, C.-Y. (2021). Ddpg-based adaptive robust tracking control for aerial manipulators with decoupling approach. *IEEE Transactions on Cybernetics*. doi: 10.1109/TCYB.2021.3049555

Lones, M. A. (2014). Metaheuristics in nature-inspired algorithms. *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 1419–1422. July 12 - 16, Vancouver, Canada. doi: 10.1145/2598394.2609841

Low, E. S., Ong, P., & Cheah, K. C. (2019). Solving the optimal path planning of a mobile robot using improved q-learning. *Robotics and Autonomous Systems*, *115*, 143–161. doi: 10.1016/J.ROBOT.2019.02.013

Lv, L., Zhang, S., Ding, D., & Wang, Y. (2019). Path planning via an improved dqn-based learning policy. *IEEE Access*, *7*, 67319–67330. doi: 10.1109/ACCESS.2019.2918703

Lynch, K. M., & Park, F. C. (2017). *Modern robotics*. Cambridge University Press.

Malik, A., Henderson, T., & Prazenica, R. (2021a). Multi-objective swarm intelligence trajectory generation for a 7 degree of freedom robotic manipulator. *Robotics*, *10*(4), 127. doi: 10.3390/robotics10040127

Malik, A., Henderson, T., & Prazenica, R. J. (2021b). Trajectory generation for a multibody robotic system using the product of exponentials formulation. *AIAA Scitech 2021 Forum*, 2016. 19–21 January, Virtual. doi: 10.2514/6.2021-2016

Malik, A., Henderson, T., & Prazenica, R. J. (2021c). Using products of exponentials to define (draw) orbits and more. *Advances in the Astronautical Sciences*, *175*, 3319.

LIST OF PUBLICATIONS

Journal Articles:

Malik, A., Henderson, T., Prazenica, R. (2021). Multi-Objective Swarm Intelligence Trajectory Generation for a 7 Degree of Freedom Robotic Manipulator. *Robotics*, *10*(4), 127. doi: 10.3390/robotics10040127

Malik, A., Lischuk, Y., Henderson, T., Prazenica, R.J. (2021). A deep reinforcement learning approach for inverse kinematics solution of a high degree of freedom robotic manipulator. *Paper is under review.*

Conference Proceedings:

Hays, C. W., Posada, D., Malik, A., Korczyk, D., Dafoe, B., Henderson, T. (2021). Structural design and impact analysis of a 1.5u cubesat on the lunar surface. *31st AAS/AIAA Space Flight Mechanics Meeting*, February 1-3, Virtual

Korczyk, J. J., Posada, D., Malik, A., Henderson, T. (2021). Modeling of an on-orbit maintenance robotic arm test-bed. *021 AAS/AIAA Astrodynamics Specialist Conference*, August 8-12, Big Sky, USA

Malik, A., Henderson, T., Prazenica, R.J. (2021). Trajectory generation for a multibody robotic system using the product of exponentials formulation. *AIAA Scitech 2021 Forum, 2016.* doi: 10.2514/6.2021-2016

Malik, A., Henderson, T., Prazenica, R.J. (2021). Using products of exponentials to define (draw) orbits and more. *Advances in the Astronautical Sciences,175, 3319.*

Malik, A., Lischuk, Y., Henderson, T., Prazenica, R.J. (2021). Generating constant screw axis trajectories with quintic time scaling for end-effector using artificial neural network and machine learning. *IEEE CCTA 2021 (accepted and presented, awaiting publishing)*

Malik, A., Lischuk, Y., Henderson, T., & Prazenica, R. J. (2021a). A deep reinforcement learning approach for inverse kinematics solution of a high degree of freedom robotic manipulator. *Paper is under review*.

Malik, A., Lischuk, Y., Henderson, T., & Prazenica, R. J. (2021b). Generating constant screw axis trajectories with quintic time scaling for end-effector using artificial neural network and machine learning. *IEEE CCTA 2021 (accepted and presented, awaiting publishing)*.

Markiewicz, B. (1973). *Analysis of the computed torque drive method and comparison with conventional position servo for a computer-controlled manipulator*. Jet Propulsion Laboratory, California Institute of Technology.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mohamed, A. Z., Lee, S. H., Aziz, M., Hsu, H. Y., & Ferdous, W. M. (2010). A proposal on development of intelligent pso based path planning and image based obstacle avoidance for real multi agents robotics system application. *2010 2nd International Conference on Electronic Computer Technology*, 128–132. 7-10 May, Kuala Lumpur, Malaysia. doi: 10.1109/ICECTECH.2010.5479974

Müller, A. (2019). An overview of formulae for the higher-order kinematics of lower-pair chains with applications in robotics and mechanism theory. *Mechanism and Machine Theory*, *142*, 103594. doi: 10.1016/J.MECHMACHTHEORY.2019.103594

Murray, R. M., Li, Z., & Sastry, S. S. (2017). *A mathematical introduction to robotic manipulation*. CRC press. doi: 10.1201/9781315136370

Pac, M. R., & Popa, D. O. (2013). Interval analysis of kinematic errors in serial manipulators using product of exponentials formula. *IEEE Transactions on Automation Science and Engineering*, *10*(3), 525–535. doi: 10.1109/TASE.2013.2263384

Paden, B. E. (1985). Kinematics and control of robot manipulators. *Ph. D. Dissertation, UNIVERSITY OF CALIFORNIA, BERKELEY,*.

Park, F. C. (1994). Computational aspects of the product-of-exponentials formula for robot kinematics. *IEEE Transactions on Automatic Control*, *39*(3), 643–647. doi: 10.1109/9.280779

Park, F. C., Kim, B., Jang, C., & Hong, J. (2018). Geometric algorithms for robot dynamics: A tutorial review. *Applied Mechanics Reviews*, *70*(1). doi: 10.1115/1.4039078

Park, J., Choi, Y., Chung, W. K., & Youm, Y. (2001). Multiple tasks kinematics using weighted pseudo-inverse for kinematically redundant manipulators. *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, *4*, 4041–4047. 21-26 May, Seoul, South Korea. doi: 10.1109/ROBOT.2001.933249

Park, K.-H., Kim, Y.-J., & Kim, J.-H. (2001). Modular q-learning based multi-agent cooperation for robot soccer. *Robotics and Autonomous Systems*, *35*(2), 109–122. doi: 10.1016/S0921-8890(01)00114-2

Pastva, T. A. (1998). *Bezier curve fitting* (Tech. Rep.). NAVAL POSTGRADUATE SCHOOL MONTEREY CA.

Paul, R. (1972). *Modelling, trajectory calculation and servoing of a computer controlled arm* (Tech. Rep.). STANFORD UNIV CA DEPT OF COMPUTER SCIENCE. doi: 10.21236/ad0785071

Phaniteja, S., Dewangan, P., Guhan, P., Sarkar, A., & Krishna, K. M. (2017). A deep reinforcement learning approach for dynamically stable inverse kinematics of humanoid robots. *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 1818–1823. July 26-28, Guangzhou, China. doi: 10.1109/ROBIO.2017.8324682

Raghavan, M., & Roth, B. (1990). Kinematic analysis of the 6r manipulator of general geometry. *International symposium on robotics research*, 314–320. August 28-31, Tokyo, Japan. doi: 10.5555/112687.112715

Raibert, M. (1978). Manipulator control using the configuration space method. *Industrial Robot: An International Journal*, *5*(2), 69–73. doi: 10.1108/eb004494

Ram, R., Pathak, P., & Junco, S. (2019). Inverse kinematics of mobile manipulator using bidirectional particle swarm optimization by manipulator decoupling. *Mechanism and Machine Theory*, *131*, 385–405. doi: 10.1016/J.MECHMACHTHEORY.2018.09.022

Robotics, R. (2017). Sawyer documentation. *Intera SDK*, Available online: https://sdk.rethinkrobotics.com/intera/Main_Page (accessed on 10 August 2019).

Rokbani, N., & Alimi, A. M. (2013). Inverse kinematics using particle swarm optimization, a statistical analysis. *Procedia Engineering*, *64*, 1602–1611. doi: 10.1016/J.PROENG.2013.09.242

Ruan, X., Ren, D., Zhu, X., & Huang, J. (2019). Mobile robot navigation based on deep reinforcement learning. *2019 Chinese control and decision conference (CCDC)*, 6174–6178. June 3-5, Nanchang, China. doi: 10.1109/CCDC.2019.8832393

Sadhu, A. K., & Konar, A. (2017). Improving the speed of convergence of multi-agent q-learning for cooperative task-planning by a robot-team. *Robotics and Autonomous Systems*, *92*, 66–80. doi: 10.1016/j.robot.2017.03.003

Sangiovanni, B., Incremona, G. P., Piastra, M., & Ferrara, A. (2020). Self-configuring robot path planning with obstacle avoidance via deep reinforcement learning. *IEEE Control Systems Letters*, *5*(2), 397–402. doi: 10.1109/LCSYS.2020.3002852

Sangiovanni, B., Rendiniello, A., Incremona, G. P., Ferrara, A., & Piastra, M. (2018). Deep reinforcement learning for collision avoidance of robotic manipulators. *2018 European Control Conference (ECC)*, 2063–2068. June 12-15, Limassol, Cyprus. doi: 10.23919/ECC.2018.8550363

Sasaki, H., Horiuchi, T., & Kato, S. (2017a). A study on vision-based mobile robot learning by deep q-network. *2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, 799–804. doi: 10.23919/SICE.2017.8105597

Sasaki, H., Horiuchi, T., & Kato, S. (2017b). A study on vision-based mobile robot learning by deep q-network. *2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, 799–804. September 19-22, Kanazawa, Japan. doi: 10.23919/SICE.2017.8105597

Selig, J. M. (2004a). *Geometric fundamentals of robotics*. Springer Science & Business Media. doi: 10.1007/978-1-4757-2484-4

Selig, J. M. (2004b). Lie groups and lie algebras in robotics. In *Computational noncommutative algebra and applications* (pp. 101–125). Springer. doi: 10.1007/1-4020-2307-3_5

Shi, X., Guo, Z., Huang, J., Shen, Y., & Xia, L. (2020). A distributed reward algorithm for inverse kinematics of arm robot. *2020 5th International Conference on Automation, Control and Robotics Engineering (CACRE)*, 92–96. September 19-20, Dalian, China. doi: 10.1109/CACRE50138.2020.9230347

Shi, Y., & Eberhart, R. C. (1999). Empirical study of particle swarm optimization. *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, *3*, 1945–1950. July 6-9, Washington, DC, USA. doi: 10.1109/CEC.1999.785511

Shin, K., & McKay, N. (1985). Minimum-time control of robotic manipulators with geometric path constraints. *IEEE Transactions on Automatic Control*, *30*(6), 531–541. doi: 10.1109/TAC.1985.1104009

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. doi: 10.1038/nature16961

Sridharan, M., & Stone, P. (2007). Color learning on a mobile robot: Towards full autonomy under changing illumination. *IJCAI*, 2212–2217.

Starke, S., Hendrich, N., Magg, S., & Zhang, J. (2016). An efficient hybridization of genetic algorithms and particle swarm optimization for inverse kinematics. *2016 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 1782–1789. December 3-7, Qingdao, China. doi: 10.1109/ROBIO.2016.7866587

Strang, G., Strang, G., Strang, G., & Strang, G. (1993). *Introduction to linear algebra* (Vol. 3). Wellesley-Cambridge Press Wellesley, MA.

Tarokh, M., & Kim, M. (2007). Inverse kinematics of 7-dof robots and limbs by decomposition and approximation. *IEEE transactions on robotics*, *23*(3), 595–600. doi: 10.1109/TRO.2007.898983

Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., ... Riedmiller, M. (2017). Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*.

Wang, H., Lu, X., Cui, W., Zhang, Z., Li, Y., & Sheng, C. (2018). General inverse solution of six-degrees-of-freedom serial robots based on the product of exponentials model. *Assembly Automation*. doi: 10.1108/AA-10-2017-122

Wen, S., Chen, J., Wang, S., Zhang, H., & Hu, X. (2018). Path planning of humanoid arm based on deep deterministic policy gradient. *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 1755–1760. December 12-15, Kuala Lumpur, Malaysia. doi: 10.1109/ROBIO.2018.8665248

Xin, J., Zhao, H., Liu, D., & Li, M. (2017). Application of deep reinforcement learning in mobile robot path planning. *2017 Chinese Automation Congress (CAC)*, 7112–7116. October 20-22, Jinan, China. doi: 10.1109/CAC.2017.8244061

Xue, X., Li, Z., Zhang, D., & Yan, Y. (2019). A deep reinforcement learning method for mobile robot collision avoidance based on double dqn. *2019 IEEE 28th International Symposium on Industrial Electronics (ISIE)*, 2131–2136. June 12-14, Vancouver, BC, Canada. doi: 10.1109/ISIE.2019.8781522

Yang, A., Chen, Y., Naeem, W., Fei, M., & Chen, L. (2021). Humanoid motion planning of robotic arm based on human arm action feature and reinforcement learning. *Mechatronics*, *78*, 102630. doi: 10.1016/J.MECHATRONICS.2021.102630

Yang, J., Wang, X., & Bauer, P. (2019). Extended pso based collaborative searching for robotic swarms with practical constraints. *IEEE Access*, *7*, 76328–76341. doi: 10.1109/ACCESS.2019.2921621

Yang, X.-S., Cui, Z., Xiao, R., Gandomi, A. H., & Karamanoglu, M. (2013). *Swarm intelligence and bio-inspired computation: theory and applications*. Newnes. doi: 10.1016/B978-0-12-405163-8.00001-6

Yang, Y., Juntao, L., & Lingling, P. (2020). Multi-robot path planning based on a deep reinforcement learning dqn algorithm. *CAAI Transactions on Intelligence Technology*, *5*(3), 177–183. doi: 10.1049/trit.2020.0024

Yip, M., & Das, N. (2019). Robot autonomy for surgery. In *The encyclopedia of medical robotics: Volume 1 minimally invasive surgical robotics* (pp. 281–313). World Scientific. doi: 10.1142/9789813232266_0010

Yiyang, L., Xi, J., Hongfei, B., Zhining, W., & Liangliang, S. (2021). A general robot inverse kinematics solution method based on improved pso algorithm. *IEEE Access*, *9*, 32341–32350. doi: 10.1109/ACCESS.2021.3059714

Zhang, F., Leitner, J., Milford, M., Upcroft, B., & Corke, P. (2015). Towards vision-based deep reinforcement learning for robotic motion control. *arXiv preprint arXiv:1511.03791*.

Zhang, W., Gai, J., Zhang, Z., Tang, L., Liao, Q., & Ding, Y. (2019). Double-dqn based path smoothing and tracking control method for robotic vehicle navigation. *Computers and Electronics in Agriculture*, *166*, 104985. doi: 10.1016/j.compag.2019.104985

Zhong, J., Wang, T., & Cheng, L. (2021). Collision-free path planning for welding manipulator via hybrid algorithm of deep reinforcement learning and inverse kinematics. *Complex & Intelligent Systems*, 1–14. doi: 10.1007/S40747-021-00366-1