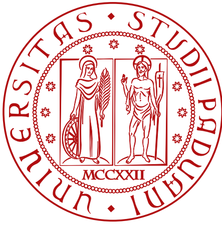
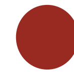


Università degli Studi di Padova



Dipartimento di Ingegneria dell'Informazione

Master Thesis in Control Systems Engineering

Trajectory Tracking Motion Optimization for Industrial Robots with One Degree of Freedom

Master Candidate

Giorgia Guglielmin

Student ID 2088623

Supervisor

Prof. Ruggero Carli

University of Padova

Co-supervisor

Dr. Paolo Scremin

Euclid Labs s.r.l.

Academic Year
2024/2025

To my family,
for their sacrifices,
and constant dedication to my happiness.

Abstract

The need for advanced optimization techniques is growing increasingly important as robotic tasks are becoming more complex, especially in industrial environments. A key focus in robotic systems is the creation of efficient and reliable trajectories that robots can execute smoothly. This thesis aims to develop feasible trajectories, that enable smooth robotic movement along designated paths, while ensuring both efficiency and reliability.
To achieve this, the research focuses on methods that prioritize energy minimization, defined as the reduction in the variation of joint angles of the robotic manipulator. By optimizing the robot's movements, unnecessary joint actions are reduced, resulting in enhanced overall efficiency and performance. The approach incorporates rotation around the Tool Axis as an additional degree of freedom, allowing the robot to handle a wider range of trajectories.
The proposed solutions include the Greedy Graph Connectivity-Based method and the Graph Energy Minimization method, both of which rely on graph construction to identify a trajectory that minimizes total energy consumption. These methods were evaluated on synthetic trajectories and applied to practical industrial scenarios such as glue application for shoe manufacturing and bicycle painting. Results indicate that the Energy Minimization approach significantly reduces unnecessary joint movements and delivers more consistent outcomes compared to the Greedy Graph method.
This work contributes to improved flexibility and efficiency in robotic motion planning within industrial settings, especially in applications where trajectory optimization is essential.

Sommario

La crescente complessità dei compiti robotici, soprattutto negli ambienti industriali, rende sempre più indispensabili tecniche di ottimizzazione avanzate. Uno tra gli aspetti fondamentali è la creazione di traiettorie efficienti e affidabili che i robot possano eseguire. Questa tesi mira a sviluppare traiettorie fattibili, che consentano un movimento robotico fluido lungo percorsi designati, garantendo al contempo efficienza e affidabilità.
Per raggiungere questo obiettivo, la ricerca si concentra su metodi che privilegiano la minimizzazione dell'energia, definita come la riduzione della variazione degli angoli dei giunti del manipolatore robotico. Ottimizzando i movimenti del robot, si riducono le azioni articolari non necessarie, con conseguente miglioramento dell'efficienza e delle prestazioni complessive. L'approccio incorpora la rotazione intorno all'asse dell'utensile come grado di libertà aggiuntivo, consentendo al robot di gestire una gamma più ampia di traiettorie.
Le soluzioni proposte includono il metodo Greedy Graph Connectivity-Based e il metodo Graph Energy Minimization, entrambi basati sulla costruzione di grafi per identificare una traiettoria che minimizzi il consumo totale di energia. Questi algoritmi sono stati valutati su traiettorie sintetiche e applicati a scenari industriali pratici, come l'applicazione di colla per la produzione di scarpe e la verniciatura di biciclette. I risultati indicano che l'approccio di minimizzazione dell'energia riduce significativamente i movimenti articolari non necessari e fornisce risultati più coerenti rispetto al metodo Greedy.
Questo lavoro contribuisce a migliorare la flessibilità e l'efficienza della pianificazione del movimento robotico in ambito industriale, soprattutto nelle applicazioni in cui l'ottimizzazione della traiettoria è essenziale.

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xvii

List of Code Snippets xvii

1 Introduction 1

1.1 Context and motivation 1
1.2 Thesis Objectives 2
1.3 EUCLID LABS 3
1.4 Thesis Structure 4

2 Theoretical Background 7

2.1 Industrial Robots 7
2.1.1 Main Components of Manipulators 8
2.1.2 Anthropomorphic Robots 9
2.2 Introduction to kinematics 10
2.3 Direct Kinematics Problems 13
2.4 Inverse Kinematics Problems 15
2.4.1 Geometric Jacobian Inverse 16
2.4.2 Newton algorithm 17
2.4.3 Gradient-descent algorithm 18
2.4.4 FABRIK Algorithm 19

3 Problem Formulation 21

3.1 Energy Function 23
3.2 State Of The Art 23

3.3 Proposed Solution . . . 25
3.3.1 Greedy Graph Connectivity-Based Approach . . . 26
3.3.2 Graph Energy Minimization Approach . . . 27
3.4 Complexity Analysis . . . 28
3.4.1 Greedy Graph Connectivity-Based Approach . . . 28
3.4.2 Graph Energy Minimization Approach . . . 30

4 Implementation 31
4.1 Synthetic Trajectories . . . 31
4.1.1 Conical Spiral Trajectory . . . 31
4.1.2 Spherical Spiral Trajectory . . . 32
4.2 Greedy Graph Connectivity-Based Approach . . . 33
4.2.1 Graph Construction Process . . . 34
4.2.2 Path Search . . . 37
4.3 Graph Energy Minimization Approach . . . 39

5 Results 43
5.1 Robot KUKA KR50 R2100 . . . 43
5.2 The User Interface . . . 44
5.3 Synthetic Trajectories . . . 45
5.4 Real-World Trajectories . . . 46
5.5 Results . . . 47
5.5.1 Greedy Graph Connectivity-Based Approach . . . 47
5.5.2 Graph Energy Minimization Approach . . . 49
5.5.3 Comparison of Approaches for Synthetic Trajectories . . . 50
5.5.4 Comparison of Approaches on Real-World Trajectories . . . 54

6 Conclusions and Future Works 55
6.1 Future Works . . . 56

References 59

Acknowledgments 61

A Appendix 63
A.1 *Denavit-Hartenberg convention* . . . 63

List of Figures

1.1 Teaching an industrial robot how to apply glue on a shoe. 2

1.2 The robotic arm with the tool direction represented in green, the tool coordinate system in blue, and the screw angle in red. 3

1.3 Company logo of EUCLID LABS 3

2.1 Examples of industrial robots from EUCLID LABS used in various applications 7

2.2 Left: Kinematic model of the human arm. Right: Robot KUKA KR 5 Arc. 10

2.3 Example of Tool reference frame 11

2.4 Description of the position and orientation of the end-effector frame 14

2.5 A full iteration of FABRIK for a single target and 4 manipulator joints, showing the movement of the end effector towards the target and the subsequent adjustments of the joints until convergence. 20

3.1 Left: tool position and orientation with respect to the tool axis. Right: experiment with β fixed at 30 degrees and α varied around the normal vector N. 22

3.2 This figure shows a Sawyer robot tracing the word "ICRA" (the purple curve is ground truth, the green curve is the robot's end effector path). STAMPEDE (right) achieves greater accuracy than RelaxedIK (left) when finding a feasible robot motion that traces the input Cartesian path. 25

3.3 Graph 27

4.1 The figure shows how the first and second levels are populated. For the first level, all possible configurations are connected to the "START" node with weight = 0. In the second level, the configurations that attach to at least one node of the previous level and have an acceptable weight (in this example, weight < 5) are calculated. 37

5.1 On the left: The KUKA KR50 R2100 robot displayed within its simulated working environment. On the right: The graphical representation of the robot's workspace. 44
5.2 User Interface 44
5.3 User interface: on the left, the imported trajectory is displayed with all trajectory points and configurations; on the right, the optimized trajectory is shown with selected configurations highlighted in green. 45
5.4 Conical spiral trajectory (100 samples) with $z_{\text{max}} = 500$ mm, $a = \frac{z_{\text{max}}}{m \cdot \text{maxRotation}}$, and $m = \cos\left(\frac{\pi}{6}\right)$. 46
5.5 Spherical spiral trajectory (100 samples) with $r = 200$ mm and ϕ limited to $\frac{\pi}{2}$. 46
5.6 Planar spiral trajectory (100 samples) with $r = 200$ mm, ϕ limited to $\frac{\pi}{2}$ and $z(\phi) = 700$mm. 46
5.7 glue application to shoes. 47
5.8 Bike painting process. 47
5.9 The `countRestart` variable in conical spiral trajectories. 48
5.10 Absolute difference in energy between two different tests ($|E_1 - E_2|$). 49
5.11 Energy results for different `maxTrajectories` values. 49
5.12 Results for the planar spiral trajectory. 51
5.13 Results for the conical spiral trajectory. 52
5.14 Results for the spherical spiral trajectory. 53

A.1 Denavit-Hartenberg kinematic parameters 64

List of Tables

5.1 Stopped levels for the conical trajectory 50
5.2 Comparison of computation time, total energy, average energy component, and variance for real-world trajectories. 54

List of Algorithms

1 **CreateGraph** method 35
2 **AddLevel** method 36
3 **FindLongestPath()** method 37
4 **DFS()** method 38
5 Energy-Based Approach 39
6 GenerateConfigurations Method 40
7 InitializeTrajectoryMinimumEnergy Method 41

List of Code Snippets

4.1 Node Class 33
4.2 Connection Class 33
4.3 GraphLevel Class 33
4.4 Graph Class 34

Introduction

1.1 Context and motivation

In today's rapidly evolving technological landscape, robotics is experiencing unprecedented expansion and transformation. More and more companies, from manufacturing and healthcare to logistics and services, are turning to automation specialists [2]. Robotic technology enhances efficiency and productivity and introduces new possibilities for innovation and improved life's quality [7].
As the complexity of robotic systems increases, the need for advanced algorithms and techniques to optimize their performance grows. A key focus in robotic systems is the creation of efficient and reliable trajectories that robots can execute smoothly. The primary goal of this thesis is to find a feasible trajectory that ensures the robot can physically move along the desired path.
The solution to the previous problem is often too restrictive, and in many real-world applications, it would be advantageous to incorporate additional degrees of freedom for the target pose. This approach allows for overcoming reachability issues, avoiding singularities, and optimizing the robot's movements, ultimately facilitating easier scheduling and reducing production times. Consider the case in which one wants to make the robot follow a trajectory obtained from the movements of an operator to mimic his movements. This data can be obtained through cameras, gyroscopes, or commercial devices such as OptiTrack or RealSense. However, the data obtained often contains critical issues for the robot, which must be avoided. Examples of such applications include welding, gluing shoes or other parts (Figure 1.1), and surface cleaning operations, in

which the main purpose is to cover the entire surface rather than maintaining a specific orientation of the cleaning tool. In those cases, the most important data is the position of the robot's end effector, while some components of the orientation may be neglected.

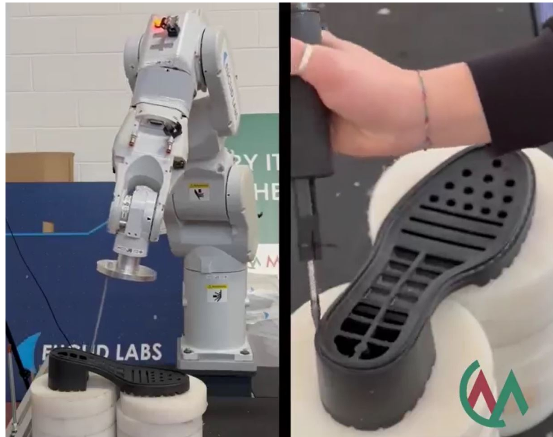


Figure 1.1: Teaching an industrial robot how to apply glue on a shoe.

1.2 Thesis Objectives

The primary motivation for this thesis is to develop a comprehensive solution that enhances the trajectory-tracking capabilities of robotic arms. The focus is on generating a feasible trajectory that minimizes the movement of the robot's joints by leveraging an additional degree of freedom in the orientation.
To achieve this, the approach allows for adjustments in the screw angle while keeping the approach direction (or tool axis) fixed with respect to the recorded position (Figure 1.2). This flexibility facilitates the generation of a feasible trajectory without compromising the tool's orientation, which is critical in tasks such as glue application or surface cleaning.

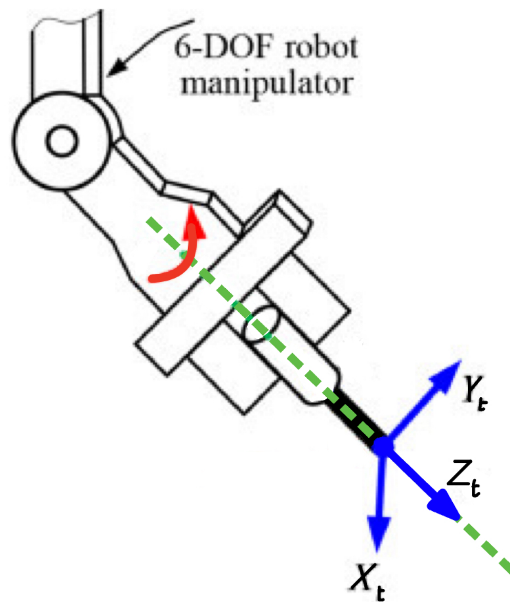


Figure 1.2: The robotic arm with the tool direction represented in green, the tool coordinate system in blue, and the screw angle in red.

1.3 EUCLID LABS

This thesis has been conducted in collaboration with EUCLID LABS, a leading company in the field of robotics and industrial automation. The logo can be seen in Figure 1.3, and the following information is obtained from the company's website [3].

Figure 1.3: Company logo of EUCLID LABS

Founded in 2005 by Matteo Peluso and Roberto Polesel, EUCLID LABS' mission is to create flexible and effective software tools that address the challenges

of small-batch manufacturing, high-quality requirements, and complex task automation. Their vision is to increase the profitability of automation by minimizing programming time and adding adaptive skills, allowing manufacturers to respond to evolving market demands efficiently. With the experience gained and the various successes, the company is expanding more and more in different countries, always providing new solutions for every sector. EUCLID LABS offers a variety of innovative products designed to enhance automation processes in industrial settings.

1.4 Thesis Structure

This thesis is organized into the following chapters:

- **Theoretical Background** (Chapter 2)
  This chapter provides an overview of industrial robotics, including fundamental concepts such as kinematics, inverse kinematics, and direct kinematics. Methods for solving these kinematic problems are also discussed.

- **Problem Formulation** (Chapter 3)
  This chapter explains the problem addressed in this thesis and presents the main idea behind the methods implemented: a greedy graph connectivity-based method and a graph energy minimization method. The chapter also includes a section on computational analysis.

- **Implementation** (Chapter 4)
  This chapter details the implementation of the greedy graph connectivity-based method and the graph energy minimization method, providing insights into the algorithms, data structures, and optimizations applied to achieve the desired results.

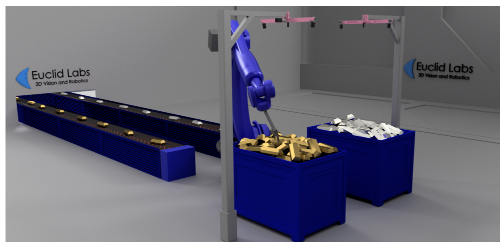
- **Results** (Chapter 5)
  This chapter presents the results obtained and provides a comparative analysis of the two methods. We analyze why the greedy method often fails to converge and discuss the improvements made to address this issue, highlighting the advantages of the energy-based method in achieving more consistent results.

- **Conclusions and Future Works** (Chapter 6)
  The final chapter summarizes the key findings of the thesis, discusses the implications of the results, and suggests potential areas for future research.

2 Theoretical Background

2.1 Industrial Robots

An industrial robot is defined as an "automatically controlled, reprogrammable multipurpose manipulator, programmable in three or more axes, which can be either fixed in place or fixed to a mobile platform for use in automation applications in an industrial environment" [**ISO8373:2021**].



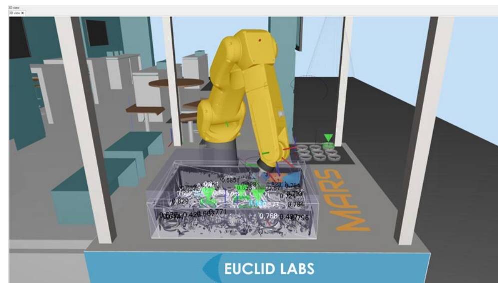


Figure 2.1: Examples of industrial robots from EUCLID LABS used in various applications

Industrial robots are designed to perform tasks with high precision and efficiency in manufacturing and production environments. Their development and application involve a combination of mechanical engineering, control theory, and computer science. Equipped with sensors, control systems, manipulators, power supplies, and specialized software, they are widely used in tasks such as:

- Assembly Operations: Thanks to their proficiency in performing complex and precise movements, these robots are ideal for assembling small and

intricate components.

- Material Handling: Their versatility allows them to handle a variety of materials — from fragile items to heavy loads — making them suitable for tasks like packaging, palletizing, and loading/unloading.
- Welding and Painting: They are well-suited for welding and painting because their precision and range of motion ensure consistent and accurate results.

Robots used in industries such as automotive assembly lines and electronics manufacturing are optimized for repetitive, precise movements. Advances in robotics have led to improvements in robot kinematics, movement optimization, and control techniques, which are crucial for enhancing performance and adaptability in industrial environments.

2.1.1 Main Components of Manipulators

By virtue of its programmability, the industrial robot is a typical component of programmable automated systems. Nonetheless, robots can be entrusted with tasks both in rigid and flexible automated systems. The main components of manipulators include [6]:

- The robot's base, typically fixed to the ground or a stationary platform, provides support and stability for the mechanical structure or manipulator, which consists of a sequence of rigid bodies (links) connected using articulations (joints). The type of joint is crucial because it determines whether the robot can perform specific movements or not. More specifically:
  - revolute joint allows rotation around a single axis and it is used in applications where the robot needs to perform rotational movements, such as bending an arm at the elbow;
  - prismatic joint, also known as a linear joint, is used to perform translational movements (for example extending or retracting a segment of the robot arm) because it allows linear movement along a single axis.

A manipulator is also characterized by an arm that ensures mobility, a wrist that confers dexterity, and an end effector that performs the task required of the robot. The end-effector is the component that interacts directly with the environment and its type depends on the specific task the robot is designed to perform.

- Actuators that set the manipulator in motion through actuation of the joints; the motors employed are typically electric and hydraulic, and occasionally pneumatic.

- Sensors that ensure the status of the manipulator (proprioceptive sensors) and, if necessary, the status of the environment (exteroceptive sensors).

- A control system (computer) that enables control and supervision of manipulator motion.

2.1.2 Anthropomorphic Robots

There are different forms of robots, each with a particular purpose, utility, and set of inherent advantages. Anthropomorphic robots, also known as articulated robots, are robots with movements on 5 or more joints, which mimic the human arm's structure and movement capabilities. This design allows for a wide range of motion, making it a significantly flexible form of automation capable of performing multiple activities across different industrial applications. As illustrated in Figure 2.2, the kinematic model of the human arm shows how these robots are designed to replicate the intricate joint movements and degrees of freedom of a human arm.

A notable example of an anthropomorphic industrial robot is the KUKA KR 5 Arc, which has a human-like arm structure with multiple rotary joints that provide a high degree of freedom similar to a human arm. This robot features advanced kinematics and control systems, enabling it to perform complex tasks with high precision. [9]

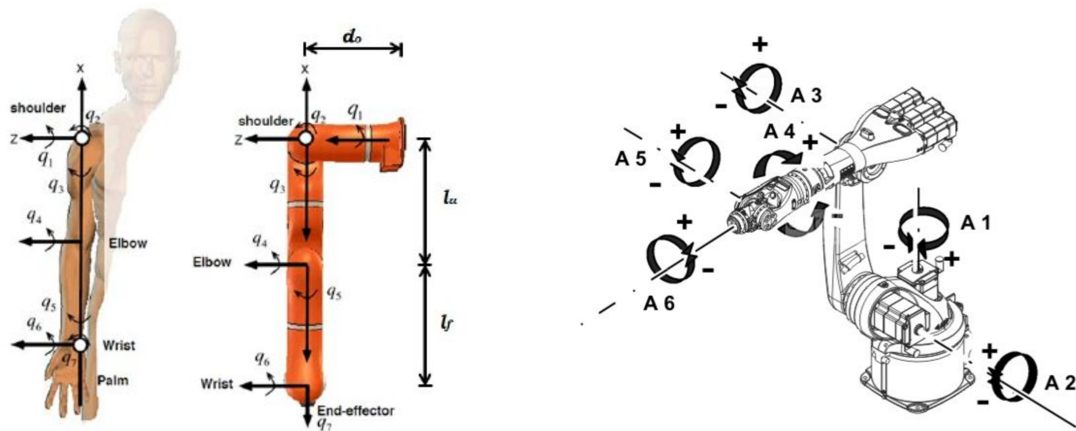


Figure 2.2: Left: Kinematic model of the human arm. Right: Robot KUKA KR 5 Arc.

2.2 Introduction to kinematics

Kinematics is a fundamental and classical topic in robotics, focusing on the relationship between a robot's joint coordinates and its spatial layout. Kinematics can provide precise calculations for tasks like positioning a gripper in space, designing a mechanism that can move a tool from point A to point B, or predicting whether a robot's motion would collide with obstacles. Kinematics focuses only on the robot's current position and ignores movement caused by forces and torques (which are covered in dynamics). The kinematics problem may be rather trivial for certain robots, like mobile robots that are essentially rigid bodies, but requires involved study for other robots with many joints, such as humanoid robots and parallel mechanisms.

Reference Frames

The frames attached to the base and the end-effector are essential for describing the robot's kinematics. The base frame is termed $O_b - x_b y_b z_b$ and provides a fixed reference point for the robot's movements. The frame attached to the end-effector allows us to define the final position and orientation of the robot's working tool. The origin of this frame is known as the Tool Center Point (TCP), i.e. the precise point where the end-effector interacts with the environment, such as the tip of a welding tool or the center of a gripper. This end-effector frame is typically defined by three unit vectors: x_e, y_e, z_e, which are represented in blue

in Figure 2.3.

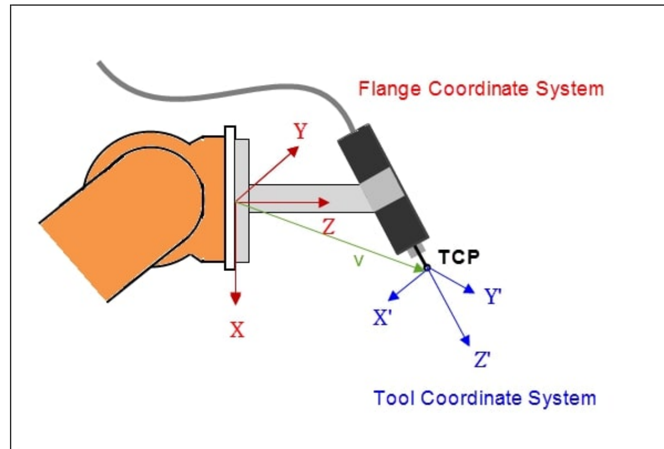


Figure 2.3: Example of Tool reference frame

In addition to understand these frames, it is crucial to define the Tool Axis, which represents the direction along which the tool interacts with the environment. The Tool Axis is aligned with the tool's working direction, adapting to the specific task requirements.
Understanding these frames and how they relate to each other is crucial for programming and controlling industrial robots.

Configuration Space, Operational Space, and Workspace

A m-dimentional vector can describe the end-effector pose

$$x_e = \begin{bmatrix} p_e \\ \phi_e \end{bmatrix} \tag{2.1}$$

where p_e is the position and ϕ_e is the orientation. This vector is defined in the space in which the manipulator task is specified (operational space) while the joint space (configuration space) denotes the space in which we define the $(n \times 1)$ vector of joint variables:

$$q = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix}$$

Each entry q_i of this vector represents the value of a joint, which can be either an angular displacement θ_i for a revolute joint, or a linear displacement d_i for

a prismatic joint. A configuration of the robot is a complete specification of the joint values, and therefore, the set of all possible configurations defines the configuration space. Knowing the values of all joint variables makes it possible to determine the position of every point on the manipulator, given that the base is fixed and the links are rigid. The number of degrees of freedom (DOF) of the robot corresponds to the minimum number of independent parameters needed to specify its configuration, which is also the dimensionality of the configuration space.

In addition to the configuration and operational space, there is also the so-called workspace: the region described by the origin of the end-effector frame when all the manipulator joints execute all possible motions. There are generally two types of workspace that are considered:

- Reachable Workspace: This is the set of all points that the origin of the end-effector frame can reach with at least one orientation.
- Dexterous Workspace: This is a subset of the reachable workspace where the end effector can reach a point while attaining any orientation. The dexterous workspace is typically smaller than the reachable workspace because the orientation constraints reduce the number of positions the end effector can achieve.

Redundancy in Robot Kinematics

Regarding the above-defined spaces, a manipulator can have different types of redundancy depending on its structure and task. These include:

- Intrinsic redundancy: if the dimension of the joint space is greater than the dimension of the operational space ($m > n$). This type of redundancy is inherent to the robot's structure: the manipulator has more DOF than the number required to fully define its position and orientation in space. *Example:* a robot with 7 DOF operating in a 3D space requires only 6 DOF (3 for position and 3 for orientation).
- Kinematic redundancy: when the number of variables necessary to describe a given task does not exceed the number of DOF. This redundancy

is task-specific: a manipulator can be redundant w.r.t one task and non-redundant w.r.t another.
Example: A 7-DOF robot arm tasked with positioning an object in space (which requires only 6 DOF) is kinematically redundant for this specific task, because positioning the object requires only 3 DOF (position in space).

- Functional redundancy: even when $m = n$, a manipulator may exhibit redundancy if only r components of the operational space are relevant to the specific task, with $r < m$.
  Example: a 6 DOF robot arm that only needs to position an object in space without worrying about its orientation. In this case, only 3 DOF are necessary to achieve the task.

Redundancy—whether intrinsic, kinematic, or functional—offers significant advantages in robotics. It allows for greater flexibility in task execution, enabling the robot to avoid obstacles, optimize joint movement, or reduce energy consumption. Multiple joint configurations can be used to achieve the same end-effector position, which helps in optimizing criteria such as minimizing joint velocities or avoiding mechanical limits.

2.3 Direct Kinematics Problems

Since both orientation and position depend on the joint variables, the direct kinematics equation can be written as

$$x_e = \kappa(q) \tag{2.2}$$

where the $(m \times 1)$ vector function $\kappa(.)$ allows the computation of the operational space variables from the knowledge of the joint space variables. In other words, in this case the aim is to compute the pose of the end-effector as a function of the joint variables. Hence with the respect to a reference frame $O_b - x_b y_b z_b$, the direct kinematics function is expressed by the homogeneous transformation matrix:

$$T_e^b(q) = \begin{bmatrix} x_e^b(q) & y_e^b(q) & z_e^b(q) & p_e^b(q) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where q is the vector of joint variables, x_e, y_e, z_e are the unit vectors of a frame attached to the end-effector, and p_e is the position vector of the origin of such a frame w.r.t. the origin of the base frame (see Figure 2.4).

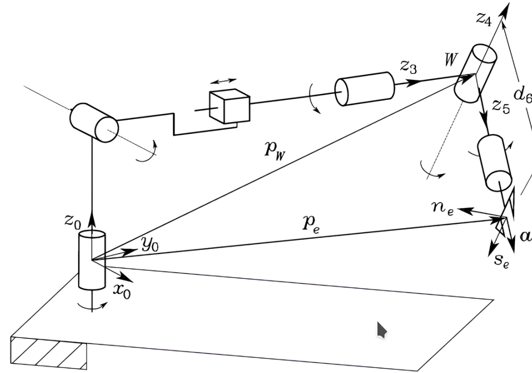


Figure 2.4: Description of the position and orientation of the end-effector frame

An operating procedure for the computation of direct kinematics can be naturally derived from the typical open kinematic chain of the manipulator structure. Since each joint connects two consecutive links, it is logical first to describe the kinematic relationship between adjacent links and then recursively derive the overall kinematic description. To this purpose, following the so-called *Denavit-Hartenberg convention* (DH), which is explained in Appendix A.1, let us define a coordinate frame attached to each link, from Link 0 to Link n. Then, the coordinate transformation describing the position and orientation of Frame n w.r.t. Frame 0 is given by:

$$T_n^0(q) = A_1^0(q_1)A_2^1(q_2)\dots A_n^{n-1}(q_n) \tag{2.3}$$

where $A_i^{i-1}(q_i)$ is the homogeneous coordinate transformation from Frame i to Frame i-1. Given T_0^b and T_e^n, compute the direct kinematics function as

$$T_e^b = T_0^b T_n^0 T_e^n \tag{2.4}$$

that yields the position and orientation of the end-effector frame with respect to the base frame.

2.4 Inverse Kinematics Problems

In most industrial applications, the Cartesian variables are provided, because the final position of the end effector defines the task requirements and the corresponding joint variables must be calculated to ensure task execution. In this case it is possible to talk about "inverse kinematics".
Let $\mathbf{x}_e^d$ be the end-effector's desired pose. We would like to solve the nonlinear equation

$$x_e^d = \kappa(q) \tag{2.5}$$

where q is the vector of the n unknowns. Solving this problem is critically important in order to transform the motion specifications, assigned to the end-effector in the operational space, into the corresponding joint space motions that allow execution of the desired motion. Concerning the direct kinematics equation, once the joint variables are known there is only one way to compute the end-effector position and rotation matrix. On the other, the inverse kinematics problem is much more complex for the following reasons:

- The equations to solve are typically nonlinear (it is not always possible to find a closed-form solution).
- There may be multiple possible solutions.
- There may be infinite solutions in the case of a kinematically redundant manipulator.
- There might be no admissible solutions depending on the manipulator's kinematic structure (for instance, if the given end-effector position does not belong to the manipulator reachable workspace).

On the other hand, the problem of multiple solutions depends not only on the number of degrees of freedom (DOF) but also on the number of non-zero DH parameters: the greater the number of non-null parameters, the greater the number of admissible solutions. For example, there are in general up to 16 admissible solutions for a six-DOF manipulator without mechanical joint limits. This leads to the need to find criteria for choosing between the admissible solutions. For the real structure, the number of admissible multiple solutions can be

reduced by the existence of mechanical joint limits.
Computation of closed-form solutions requires :

- algebraic intuition to identify the key equations involving the unknowns.
- geometric intuition to find those relevant points on the structure with respect to which it is convenient to express position and/or orientation as a function of a reduced number of unknowns.

In many practical cases, finding closed-form solutions is either impossible or extremely difficult and it may be necessary to resort to numerical solution techniques (like Newton algorithm or Gradient-descent algorithm); these have the advantage of being applicable to any kinematic structure, but in general they do not allow computation of all admissible solutions.

2.4.1 Geometric Jacobian Inverse

Differential kinematics gives the relationship between the joint velocities and the end-effector's linear and angular velocity. This mapping is described by a matrix, known as the geometric Jacobian $J(q)$, which depends on the manipulator configuration.
More specifically, we can express the end-effector's linear velocity $\dot{p}_e$ and angular velocity ω_e as functions of the joint velocities $\dot{q}$:

$$v_e = \begin{bmatrix} \dot{p}_e \\ \omega_e \end{bmatrix} = \begin{bmatrix} J_P(q) \\ J_O(q) \end{bmatrix} \dot{q} \tag{2.6}$$

where both J_P and J_O are $(3 \times n)$ matrices. This equation establishes a linear mapping between the joint velocity space and the operational velocity space, though it depends on the current configuration. This fact suggests that the differential kinematics equation could be used to address the inverse kinematics problem.
Assume that a motion trajectory is assigned to the end-effector in terms of v_e and the initial conditions on position and orientation. The goal is to find a joint trajectory $(q(t), \dot{q}(t))$ that can reproduce the given trajectory. By considering Equation 2.6, the joint velocities can be obtained by inverting of the Jacobian matrix

$$\dot{q} = J^{-1}(q) v_e. \tag{2.7}$$

If $q(0)$ is known, the positions can be computed by integrating velocities over time, i.e.,

$$q(t) = \int_0^t \dot{q}(\eta)\, d\eta + q(0). \tag{2.8}$$

This integration process can be executed in discrete time by using numerical techniques. The simplest technique is based on the Euler integration method; given a time step Δt, if the joint positions and velocities at time t_k are known, the joint positions at the subsequent time $t_{k+1} = t_k + \Delta t$ are

$$q(t_{k+1}) = q(t_k) + \dot{q}(t_k)\Delta t. \tag{2.9}$$

This technique for inverting kinematics does not rely on the kinematic structure's solvability. Nonetheless, the Jacobian must be square and of full rank, otherwise the direct inversion of $J(q)$ is not feasible. To address this, in many practical applications, the pseudoinverse of the Jacobian, denoted as $J^{\dagger}(q)$, is implemented. The pseudoinverse provides a least-squares solution that minimizes the error in the operational space, ensuring a feasible joint trajectory even in the presence of redundancy or singularities.
Equation 2.7 can be rewritten as

$$\dot{q} = J^T(JJ^T)^{-1}v_e = J^{\dagger}(q)v_e. \tag{2.10}$$

Using the pseudoinverse is particularly advantageous when dealing with redundant manipulators, where several joint configurations can reach the same end-effector pose. In these scenarios, the pseudoinverse allows for the selection of a solution that optimizes a secondary criterion, such as minimizing joint velocities or avoiding joint limits.

2.4.2 Newton algorithm

Consider the case where $m = n$. The main idea is to generate a sequence of values for q, starting from an initial guess q^0 that, hopefully, will converge to a solution $\mathbf{q}^*$ of Equation 2.5.
Let q_k be the value the joint variable vector attained at the k-th iteration. By computing the first-order Taylor expansion of Equation 2.5 around q^k, we obtain

$$x_e^d = \kappa(q) \approx \kappa(q^k) + J_A(q^k)(q - q^k) \tag{2.11}$$

where $J_A = \frac{\partial \kappa(q^k)}{\partial q}$ is the so-called analytical Jacobian. Finally q^{k+1} is obtained by solving the equation

$$x_e^d = \kappa(q^k) + J_A(q^k)(q^{k+1} - q^k) \tag{2.12}$$

that yields

$$q^{k+1} = q^k + J_A^{-1}(q_k)\left(x_e^d - \kappa(q^k)\right). \tag{2.13}$$

Newton's method exhibits quadratic convergence when near to a solution q^*. However, convergence is not always guaranteed; the choice of the initial guess q^0 is crucial for determining it. Additionally, problems may arise when computing $J_A^{-1}(q^k)$ if q^k is close to singularities of the Jacobian matrix J_A. Finally, notice that Equation 2.13 can be applied only when $n = m$; in the case of a redundant manipulator, where $n > m$, it must be adjusted accordingly.

2.4.3 Gradient-descent algorithm

Let consider the error function

$$H(q) = \frac{1}{2}\|x_e^d - \kappa(q)\|^2$$

The main idea is to move along the direction of the negative gradient. From

$$\nabla_q H(q) = -J_A^T(q)\left(x_e^d - \kappa(q)\right)$$

we obtain

$$q_{k+1} = q^k + \alpha J_A^T(q^k)\left(x_e^d - \kappa(q^k)\right)$$

where the step size $\alpha > 0$ must be carefully chosen to ensure that the error function $H(q)$ decreases along the trajectory generated by the algorithm, meaning that $H(q_{k+1}) < H(q_k)$, as long as q_k is not a stationary point.
The gradient method is computationally simpler than the Newton method because it requires the transpose of the Jacobian matrix rather than its inverse. This makes the gradient algorithm suitable also for non-redundant manipulators. However, the algorithm may get stuck at a point q where the error $e = x_e^d - \kappa(q)$ is not zero but belongs to the kernel of $J_A^T(q)$.

2.4.4 FABRIK Algorithm

After exploring Jacobian-based and numerical methods, heuristic alternatives are worth considering to solve the inverse kinematics problem. Heuristic methods are often preferred in contexts where speed and simplicity of implementation are crucial, even at the cost of rigorous precision or ensuring an optimal solution in every case. Among these methods, FABRIK (Forward And Backward Reaching Inverse Kinematics) is one of the best-known and appreciated.

FABRIK, as explained in Fabrik [1], was introduced by Andreas Aristidou and Joan Lasenby in 2011 as an innovative method that avoids the complex matrix manipulations and singularity problems typical of Jacobian-based solutions. FABRIK stands out for its intuitive approach: instead of using rotation angles or complex matrix operations, it calculates the position of each joint by locating a point on a line between the previous and next joint. This simplified approach makes the algorithm extremely efficient and easily implementable.

FABRIK algorithm consists of two steps which can be clearly visualized in Figure 2.5:

- Forward Reaching Phase: From the end effector, FABRIK updates the joint positions by proceeding along the chain until it reaches the root joint. During this phase, each joint is repositioned along the line that connects it to the next joint, keeping the distance between adjacent joints constant.

- Backward Reaching Phase: Once the root is reached, the algorithm proceeds in reverse, repositioning the joints until returning to the end-effector. This double iterative process ensures that the solution quickly converges to a configuration where the end effector comes as close as possible to the desired target position.

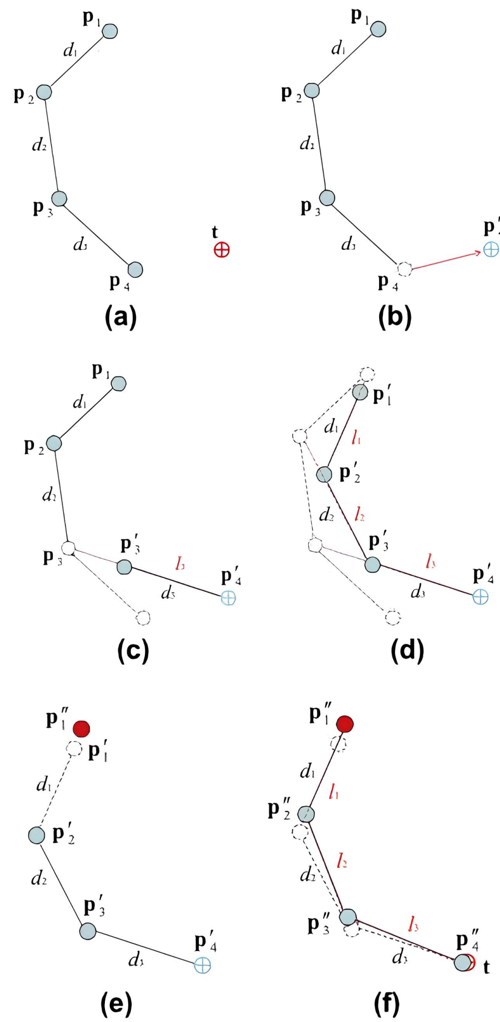


Figure 2.5: A full iteration of FABRIK for a single target and 4 manipulator joints, showing the movement of the end effector towards the target and the subsequent adjustments of the joints until convergence.

In 2020, FABRIK was extended to mobile manipulators in the M-FABRIK approach [5]. This extension, developed by Phillipe Cardoso Santos et al., was designed specifically for real-time applications with industrial robots, such as the Kuka YouBot. M-FABRIK retains the core strengths of FABRIK—low computational cost and simplicity—while introducing improvements to handle the redundancy and mobility of industrial robots mounted on mobile platforms. It has proven effective in avoiding obstacles, managing joint limits, and increasing manipulability, making it a valuable tool for industrial applications.

3

Problem Formulation

The problem addressed involves generating a sequence of joint configurations that accurately trace a given Cartesian-space trajectory while minimizing the movement of the robot's joints.
As explained in Section 2.4, the Inverse Kinematics process determines the joint angles required to achieve each pose in the trajectory and, due to the redundancy of robotic systems, multiple joint configurations can reach each pose. The challenge of this thesis is not only to solve inverse kinematics for each pose individually but also to select configurations that minimize joint movements between consecutive poses, reducing mechanical wear and energy consumption.
To quantify the movement between two configurations, the distance d between two configurations is computed as follows:

$$d = \sqrt{\sum_{i=1}^{n} (Conf_{1,i} - Conf_{2,i})^2} \tag{3.1}$$

where

$$Conf_1 = \begin{bmatrix} q_{1,1} \\ q_{1,2} \\ \vdots \\ q_{1,n} \end{bmatrix}, \quad Conf_2 = \begin{bmatrix} q_{2,1} \\ q_{2,2} \\ \vdots \\ q_{2,n} \end{bmatrix},$$

and q_i is the joint angle for a given joint i. This formula calculates the Euclidean distance between the two sets of joint angles, effectively measuring how

much the robot's joints move from one configuration to the next. Minimizing this distance between consecutive configurations is key to reducing unnecessary joint movements.
When the robot encounters situations where joint movements are excessively large or certain poses are unreachable due to workspace limitations or singularities, an additional degree of freedom is introduced: a rotation around the Tool Axis, as defined in Section 2.2. In fact, by adjusting the orientation of the tool through this rotation, the robot can explore alternative configurations to ensure that all poses in the trajectory remain reachable, while minimizing unnecessary joint movements.
In this context, this degree of freedom corresponds to a rotation around the z-axis of the end-effector's reference frame. This degree of freedom is implemented by controlling the tool attitude, which is represented by a tool axis vector T. As illustrated in Figure 3.1, the tool attitude is defined by two angles: the tool axis inclination direction angle α and the tool inclination angle β. These angles describe the tool's orientation relative to the normal vector N of the tool's working surface. In this thesis we set $\beta = 0$ while α is varied from 0 to 360 degrees around N to adjust the tool's orientation.

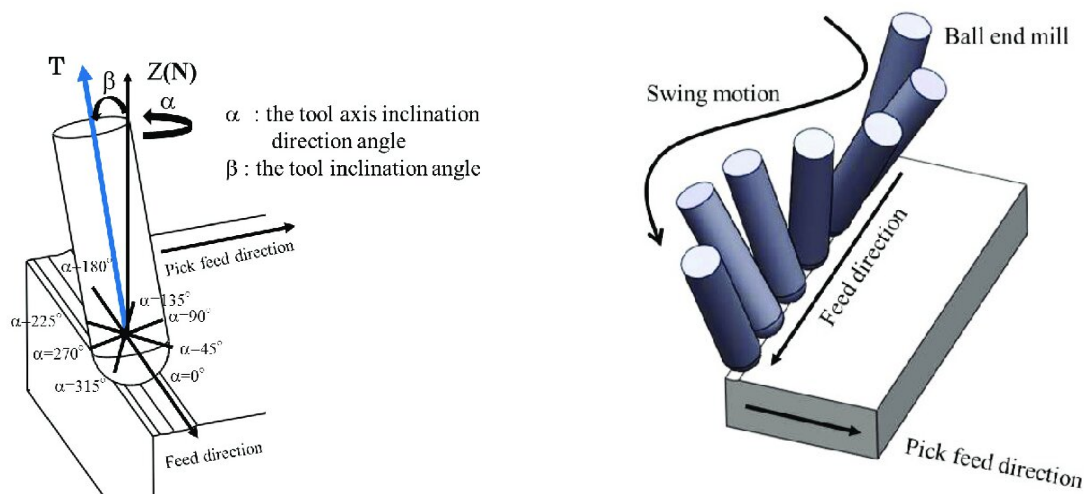


Figure 3.1: Left: tool position and orientation with respect to the tool axis. Right: experiment with β fixed at 30 degrees and α varied around the normal vector N.

Therefore, the solution proposed focuses on generating feasible trajectories by minimizing joint movements across consecutive poses, while leveraging the rotation around the Tool Axis as an additional degree of freedom. This approach helps address reachability issues, avoid singularities, and optimize the robot's

movements, making scheduling easier and improving production times.

3.1 Energy Function

We need a mathematical way to measure the quality of a trajectory in terms of feasibility. We choose the concept of energy as a metric to compare multiple trajectories:

- Reachability check: if any pose in a trajectory is not reachable (as discussed in the previous chapter), the entire trajectory is considered non-feasible, and the energy is infinite.
- Joint difference using Euclidean distances (Formula 3.1):

$$E = \sum_{j=0}^{t-1} \sqrt{\sum_{i=1}^{n} (Conf_{j,i} - Conf_{j+1,i})^2} \tag{3.2}$$

  Where:

  - t is the total number of trajectory points,
  - n is the number of the robot's joints,
  - $Conf_{j,i}$ and $Conf_{j+1,i}$ represent the robot's configurations at consecutive trajectory points, where j denotes the trajectory point and i refers to the joint.

  This formula essentially measures the overall change in the robot's joint angles, giving an indication of how much adjustment is needed to move from one point to the next.

Therefore, the goal of the thesis can be redefined as minimizing this energy.

3.2 State Of The Art

In this section, we present the STAMPEDE method [4], which addresses the problem of trajectory tracking motion optimization.

STAMPEDE is a discrete-optimization technique for finding feasible robot arm trajectories that pass through provided 6-DOF Cartesian-space end-effector paths with high accuracy, the so-called pathwise-inverse kinematics problem. The output consists of a path function of joint angles that best follows the provided end-effector path function, given some definition of "best".
The strategy used is to cast the robot motion translation problem as a discrete-space graph-search problem: each node in the graph represents a feasible configuration at a given time step, while edges between nodes represent feasible transitions between configurations. The method employs various sampling techniques such as diversity sampling, to ensure that a wide variety of configurations are considered at each step, and adaptive sampling, to refine the search space based on proximity to singularities or unreachable points.
This approach offers significant potential for use in various robotics applications, such as teaching-by-demonstration, where a user's hand motion is remapped to a feasible robot path that matches what the user's hand did with high accuracy. Additionally, the method could be used to retarget motions between different robots. Since robots have vastly different scales and capabilities in joint space, having a method that provides correspondence between robot arms in a common end-effector space can serve as a bridge to compile motions and actions between robots.
However, the method also presents some limitations. While it performs efficiently for a global optimization technique, it is still slower than local, greedy methods. Furthermore, the current framing of the pathwise-IK problem only considers static input paths and future extensions are needed to enable dynamic adjustments using incremental path search algorithms. These adaptations could allow for real-time branching and modification of paths. Lastly, the method does not consider dynamics, which could be addressed by integrating a time-scaling approach to improve its applicability in more complex scenarios.

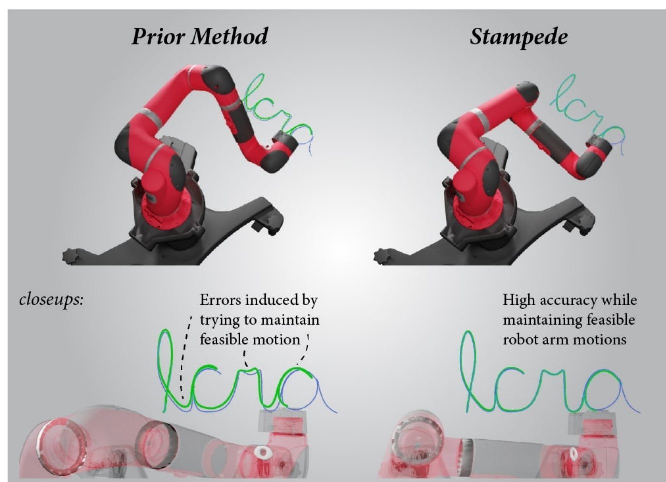


Figure 3.2: This figure shows a Sawyer robot tracing the word "ICRA" (the purple curve is ground truth, the green curve is the robot's end effector path). STAMPEDE (right) achieves greater accuracy than RelaxedIK (left) when finding a feasible robot motion that traces the input Cartesian path.

3.3 Proposed Solution

In this section we introduce two graph-based solutions to solve the trajectory optimization problem. The implementation details of each part can be found in the next chapter.

In mathematics and computer science, a graph is a structure that consists of a set of nodes (also called vertices) and edges that connect pairs of nodes. Graphs are widely used to represent relationships between entities and can model a variety of problems, from social networks to optimization issues. The approach proposed in this thesis draws inspiration from previous graph-based methods, such as STAMPEDE method, which also address inverse kinematics and path-planning problems using a graph-search strategy, as detailed in Section 3.2.
Both proposed solutions use a graph-based method in which the trajectory points are discretized into nodes, each representing a possible joint configuration that allows the robot to reach a specific pose. The connections between nodes indicate feasible transitions between configurations at consecutive trajectory points. Since the edges can only be traversed in one direction and there are no cycles (no paths lead back to a previously visited node), the resulting structure is a Directed Acyclic Graph (DAG). These edges are weighted based on the Euclidean

distance between the configurations, as defined in Equation 3.1.

3.3.1 Greedy Graph Connectivity-Based Approach

The main idea is to construct the graph dynamically, adding feasible nodes to the graph as they are found. We evaluate each pose in sequence, ensuring that the trajectory remains valid up to that point. Once the graph is created, we proceed by searching for a possible path through the graph, meaning one that connects the first configuration to the last.
This greedy approach makes adjustments as issues arise. For this reason, it does not guarantee that there isn't another trajectory with less joint movement, as the algorithm stops once it finds a feasible configuration.

Key Steps:

1. Graph Construction Process:
   For each trajectory point, all possible joint configurations are calculated by using the Inverse Kinematics process. These nodes are only added to the graph if they have feasible connections with the nodes from the previous trajectory point (level). If no feasible connections are found, the robot's pose is rotated around the Tool Axis a certain number of times to explore alternative configurations that may be reachable or that require smaller joint movements. If, after multiple rotations, no valid connections are identified, the process becomes recursive, revisiting and recalculating configurations at the previous levels to attempt new connections.
   A representation of the final graph is shown in Figure 3.3.

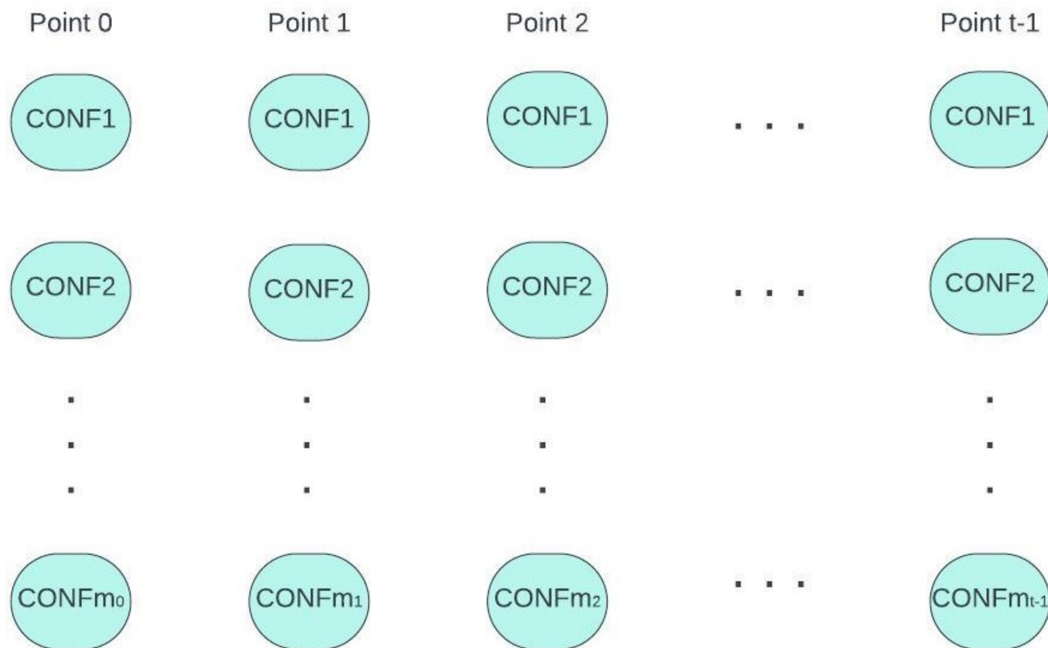


Figure 3.3: Graph

2. Path Search:
   After constructing the entire graph, the next step is to determine the longest feasible path with the minimum weight from the starting configuration to the final configuration. This longest path is identified using a Depth-First Search algorithm, which explores all possible paths through the graph and selects the one with the maximum number of nodes (configurations).
   DFS [8] is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node) and explores as far as possible along each branch before backtracking.

3.3.2 Graph Energy Minimization Approach

This approach presents an alternative to the graph-based method discussed in the previous section. The main idea is to work in the joint space: instead of rotating trajectory points dynamically whenever a problem arises, this method

precomputes all possible configurations for each pose and adds them to the dedicated graph level. These pre-generated configurations are then used to generate multiple possible trajectories, each aimed at minimizing the total energy. The algorithm generates multiple trajectories, evaluates each one based on its energy, and selects the one with the lowest overall energy as the final solution.

Key Steps:

1. Initial Configuration Generation:

   The algorithm begins by generating all feasible joint configurations for each pose along the trajectory. In other words, for each trajectory point, a pre-generated list of possible joint configurations is created. These configurations are obtained by rotating each pose around the Tool Axis and storing all valid solutions.

2. Energy-Based Selection:

   After generating configurations, the algorithm selects a set of trajectories using a process that prioritizes energy minimization. In particular, we iterate over the configurations, selecting the most energy-efficient joint transitions between consecutive poses.

3. Final Trajectory Selection:

   Once the possible trajectories are initialized, they are evaluated and ranked by the total energy required. The trajectory with the minimum energy is selected as the final solution.

3.4 Complexity Analysis

3.4.1 Greedy Graph Connectivity-Based Approach

The complexity analysis considers both the graph construction process and the path search for finding the longest path.

Best Case Scenario

- Graph Construction Process
  The algorithm processes each trajectory point sequentially, passing through each level only once without any retries or backtracking.As a result, it performs a linear traversal of the trajectory points.

- Path Search
  In the best case, each level has only one configuration, which results in the DFS algorithm having a complexity proportional to the number of trajectory points.

Overall, the time complexity for the best-case scenario can be expressed as:

$$\text{Total Complexity} : O(N) \tag{3.3}$$

with N being the number of trajectory points.

Worst Case Scenario

- Graph Construction Process
  When a trajectory point cannot be connected, the algorithm must backtrack and retry previous points. The total number of retry attempts across all levels can be expressed as:

  $$O\left(N + \sum_{i=1}^{N} R_i\right) \tag{3.4}$$

  where R_i is the number of retries for level i. Since retries are finite and bounded by a constant, the complexity remains $O(N)$.

- Path Search
  The algorithm includes a depth-first search (DFS) for finding the longest path in the constructed graph. The DFS has a time complexity of:

  $$O(N \times 8 + N \times 8^2) \approx O(N) \tag{3.5}$$

  where:

- $N \times 8$ is the number of nodes, with 8 being the average number of configurations per point.
- $N \times 8^2$ is the number of edges.

The overall time complexity can be represented as:

$$\text{Total Complexity} : O(N) + O(N) \approx O(N) \tag{3.6}$$

with N being the number of trajectory points.

3.4.2 Graph Energy Minimization Approach

The complexity analysis for this algorithm considers both the configuration generation process and the evaluation of energy between trajectory.
For each trajectory point, the algorithm generates configurations by rotating the points around the tool axis and checking for valid solutions. In the worst case, this process results in $(P \times 8)$ configurations per level, where P represents the number of generated rotation points, and 8 is the number of potential solutions per rotation. Subsequently, the algorithm iterates over the configurations to find the trajectory with minimum energy. This involves evaluating all combinations of configurations, which has a time complexity proportional to $(N \times (P \times 8)^2 \times \texttt{max_Trajectories})$, where `max_Trajectories` denotes the maximum number of feasible trajectories considered per point.

The overall time complexity can be represented as:

$$\text{Total Complexity} : O(N \times (P \times 8)^2 \times \texttt{max_Trajectories}) \approx O(N) \tag{3.7}$$

with N being the number of trajectory points.

4

Implementation

This chapter outlines the implementation of the methods introduced in Chapter 3 : Section 4.1 explains how the trajectories are generated, while Section 4.2 and Section 4.3 describe the implementation of the Greedy Graph Connectivity-Based Approach and the Graph Energy Minimization Approach, respectively.

4.1 Synthetic Trajectories

Trajectory generation involves calculating the series of poses (positions and orientations) that a robot's end-effector must follow to complete a desired motion. In the following sections, the trajectories used in this thesis are described: a conical spiral and a spherical path.

4.1.1 Conical Spiral Trajectory

The conical spiral trajectory is generated by simulating a mathematical curve known as a conical spiral, which is a three-dimensional spiral wrapped around a cone. The primary parameters defined for this trajectory include the number of sample points and the maximum height (z_{Max}). These parameters allowed the trajectory to be fine-tuned according to the specific requirements of the task. The equations mathematically define the conical spiral trajectory:

$$
\begin{aligned}
x(t) &= a \cdot \theta \cdot \cos(\theta) \\
y(t) &= a \cdot \theta \cdot \sin(\theta) \\
z(t) &= -m \cdot a \cdot \theta
\end{aligned}
\tag{4.1}
$$

where t represents the sample point index, θ is the rotation angle, a is a scaling factor derived from the geometric properties of the cone and m is the slope of the cone's lines with respect to the x-y plane.

The trajectory points are computed iteratively within a loop, where each iteration calculates the next position in 3D space using the above equations. For each point, in addition to the position, a corresponding orientation matrix is generated: x-axis is tangent to the cone's surface, z-axis normal to the cone's surface and finally y-axis as a vector product between x and z.

4.1.2 Spherical Spiral Trajectory

The spherical trajectory involved generating points on the surface of a sphere by following a procedure very similar to that used for the cone in the previous section.
The spherical trajectory is defined using spherical coordinates, which are converted into Cartesian coordinates for the robot's reference frame. The equations used are:

$$
\begin{aligned}
x(\phi, \theta) &= r \cdot \sin(\phi) \cdot \cos(\theta) \\
y(\phi, \theta) &= r \cdot \sin(\phi) \cdot \sin(\theta) \\
z(\phi) &= r \cdot \cos(\phi)
\end{aligned}
\tag{4.2}
$$

where ϕ is the polar angle, θ is the azimuthal angle, and r is the radius of the sphere.
Points on the sphere are calculated in a loop, with the polar and azimuthal angles incremented in each iteration to ensure even distribution across the sphere's surface. The orientation is set in the same way as described in Section 4.1.1.

4.2 Greedy Graph Connectivity-Based Approach

The first step is to construct a graph, which is built and managed using the following key classes: `Node`, `Connection`, `GraphLevel`, and `Graph`.

• **Node Class:** Represent a specific robot configuration at a trajectory point.

```
class Node:
    % Attributes
    List<Node> Parents
    List<Connection> ChildConnections
    double[] RobotInternalAxes

    % Method
    def TryAttach(solution: double[]):
        % Check if the connection is feasible (the weight is acceptable)
        return True if the connection is feasible, False otherwise
```

Code 4.1: Node Class

• **Connection Class:** Represent a connection between two nodes, including the weight of the transition.

```
class Connection:
    % Attributes
    Node ChildNode
    double Weight

    % Method
    def CalculateWeight(joints1: double[], joints2: double[]):
        % Returns the weight between two joint configurations (Equation 3.1)
        return computedWeight
```

Code 4.2: Connection Class

• **GraphLevel Class:** Represent a single level in the graph, containing multiple nodes.

```
class GraphLevel:
    % Attributes
    List<Node> Nodes
```

```
4     elTrajectoryElement TrajectoryElement
```

Code 4.3: GraphLevel Class

- **Graph Class:** Manage the overall graph construction and pathfinding process. This class plays a crucial role in the process of constructing the graph by using the `Node`, `Connection`, and `GraphLevel` classes.

```
1 class Graph:
2
3     % Attribute
4     GraphLevel[] Levels
5
6     % Methods
7     elTrajectory CreateGraph(elTrajectory traj, elRobot robot)
8         % Generate the graph for the given trajectory and robot
9
10    void AddLevel(int level, elTrajectory trj, elRobot robot)
11        % Add a new GraphLevel to the graph
12
13    List<Node> FindLongestPath()
14        % Find the longest path through the graph
15
16    void DFS(Node current, Set visited, List<Node> currentPath, List<
      Node> longestPath)
17        % Depth-First Search to explore the graph and find paths
```

Code 4.4: Graph Class

4.2.1 Graph Construction Process

The `CreateGraph` method (shown in Algorithm 1) manages the entire process by calling `AddLevel` to add each new level of nodes to the graph as it is computed. When all levels have been filled, the `FindLongestPath()` method (Section 4.2.2) returns the selected final path.

Algorithm 1 CreateGraph method

```
1:  Input: elTrajectory traj, elRobot robot
2:  Output: elTrajectory trj
3:
4:  def CreateGraph(traj, robot):
5:      trj = traj.Clone() % Clones the input trajectory
6:      Levels = new GraphLevel[trj.Count] % Initializes levels for each trajectory point
7:      for l = 0 to trajectory.Count % Loop through each trajectory point
8:          AddLevel(l, trajectory) % Adds each level to the graph
9:      FindLongestPath() % Finds the longest valid path through the graph
10: return trj % Returns the final trajectory
```

Regarding the `AddLevel` method (reported in Algorithm 2), we first check if the pose is reachable. If it is, we continue with that pose; otherwise, we randomly rotate the pose along the z-axis until it becomes reachable.
Then we proceed differently depending on the level we are at:

1. **Level 0:** Add a special node called "START". Calculate the possible configurations (nodes) to reach pose 0 and add them with weight = 0.

2. **From Level 1 to Level t-1:** Compute all possible configurations. Each node is validated by checking if it can connect to any nodes in the previous level. If a valid connection is found, the node is added to the current level. If not, the trajectory pose is adjusted randomly , and the validation process is repeated until the maximum number of iterations is reached.
   If valid connections cannot be established after several attempts, the algorithm recursively adjusts the trajectory at previous levels to find a feasible path forward.

Algorithm 2 AddLevel method

```
1:  Input: int level, elTrajectory trj, elRobot robot
2:  Output: void
3:
4:  def AddLevel(int level, elTrajectory trj, elRobot robot):
5:    elTrajectoryElement element = trj.TrajectoryElements[level]
6:    if (element is reachable):
7:      Compute all possible configurations (validSolutions)
8:    else:
9:      trj.TrajectoryElements[level] = randomized
10:
11:   if (level == 0):
12:     Levels[level] = new GraphLevel(validSolutions)
13:   else:
14:     attachedAtLeastOnce = false
15:     iter = 0
16:     do:
17:       iter++
18:       foreach (Node node in Levels[level-1].Nodes):
19:         foreach (double[] solution in validSolutions):
20:           if (node.TryAttach(solution)):
21:             attachedAtLeastOnce = true
22:             newNode = new Node(RobotInternalAxes = solution)
23:       if (!attachedAtLeastOnce):
24:         trj.TrajectoryElements[level] = randomized
25:     while (!attachedAtLeastOnce and iter < maxIter):
26:     if (!attachedAtLeastOnce):
27:       trj.TrajectoryElements[level-1] = randomized
28:       AddLevel(level-1, trj, robot)
29:     else:
30:       Add all computed nodes in the current level
```

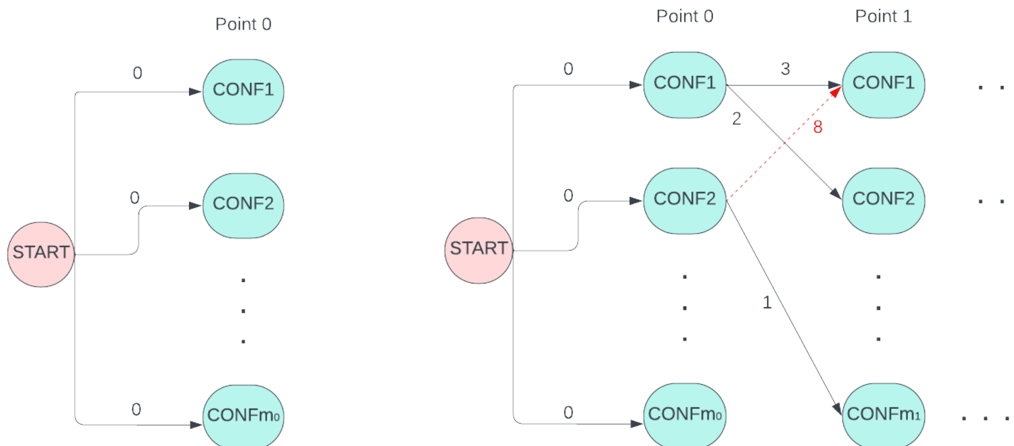


Figure 4.1: The figure shows how the first and second levels are populated. For the first level, all possible configurations are connected to the ”START” node with weight = 0. In the second level, the configurations that attach to at least one node of the previous level and have an acceptable weight (in this example, weight < 5) are calculated.

4.2.2 Path Search

To find a feasible path, the following methods are implemented:

1. **FindLongestPath()**

    - Finds the longest possible path from the start node to the end node.
    - Uses a Depth-First Search (DFS) to explore all possible paths and selects the one with the highest number of nodes.

Algorithm 3 FindLongestPath() method

1: **Input:** None
2: **Output:** List<Node> longestPath
3:
4: **def FindLongestPath():**
5: longestPath = empty list
6: visited = empty set
7: **for** level **in** Levels:
8: **for** node **in** level.Nodes:
9: currentPath = empty list
10: **DFS**(node, visited, currentPath, longestPath) % Perform depth-first search to update longestPath
11: **return** longestPath

2. **DFS(current, visited, currentPath, longestPath)**
   Below, the specific steps of how DFS algorithm works in practice are explained:

   - Support function for Depth-First Search.
   - Adds the current node to the current path and marks the node as visited.
   - If the current node has unvisited children, it explores them recursively.
   - If the current node is a leaf (without unvisited children) and the current path is longer than the longest path found so far, it updates the longest path.
   - Removes the last node from the current path and marks the node as unvisited to allow further exploration.

Algorithm 4 DFS() method

```
1:  Input: Node current, Set visited, List<Node> currentPath, List<Node> longestPath
2:  Output: None
3:
4:  def DFS(current, visited, currentPath, longestPath):
5:    add current to visited
6:    add current to currentPath
7:    hasChildren = false
8:    for connection in current.ChildConnections:
9:      child = connection.ChildNode
10:   if child not in visited then:
11:       hasChildren = true
12:       DFS(child, visited, currentPath, longestPath)
13:
14:   if !hasChildren and currentPath.length > longestPath.length then:
15:     longestPath = currentPath.clone()
16:   remove last element from currentPath
17:   remove current from visited
```

4.3 Graph Energy Minimization Approach

The core of the energy-based selection process lies in the second phase, which is managed by the `InitializeTrajectoryMinimumEnergy` method. It generates multiple trajectories for the robot by selecting configurations that minimize the weight between consecutive poses.
The method loops through a set number of initial configurations for the first point of the trajectory, as controlled by the variable `maxTrajectories`. For each initial configuration, it starts constructing a trajectory by following the next steps:

- For each subsequent pose, it evaluates the weight required to transition from the current configuration to each possible configuration of the next pose. The weight is calculated as the difference in joint angles between the configurations using `Connection.CalculateWeight()`.
- The configuration that results in the minimum weight is selected as the next one in the trajectory.
- This process repeats until a complete trajectory is generated, ensuring the weight is minimized for every transition.

The algorithm generates a fixed number of trajectories, equal to the parameter `maxTrajectories`, each starting from a different configuration for the first point. These trajectories are stored in a list for later evaluation to select the one with the lowest energy.

Algorithm 5 Energy-Based Approach

```
1:  Input: elTrajectory trj (initial trajectory)
2:  Output: elTrajectory bestTraj, List<double[]> bestConf
3:
4:  def Main(trj):
5:      % 1. Initialization
6:      allConfigurations = GenerateConfigurations(trj)
7:      % 2. Energy-Based Selection
8:      selectedConf = InitializeTrajectoryMinimumEnergy(allConfigurations)
9:      % 3. Final Trajectory Selection
10:     orderedConf = Sort(selectedConf) by energy
11:     bestConf = orderedConf[0]
12:     bestTraj = DirectKinematics(bestConf)
13: return (bestTraj, bestConf)
```

Algorithm 6 GenerateConfigurations Method

```
 1: Input: elTrajectory trj
 2: Output: List<List<double[]» allConfigurations
 3:
 4: def GenerateConfigurations(trj):
 5: for each element in trj do
 6:     configurations = []
 7:     for angle from 0 to 359 with step 1 do
 8:         rotatedElement = RotateAroundZ(element, angle)
 9:         if rotatedElement is valid then
10:             solutions = GetConfiguration(rotatedElement)
11:             if solutions are valid then
12:                 configurations.Add(solutions)
13:             end if
14:         end if
15:     end for
16:     allConfigurations.Add(configurations)
17: end for
18: return allConfigurations
```

Algorithm 7 InitializeTrajectoryMinimumEnergy Method

```
1: Input: List<List<double[]» allConfigurations, int maxTrajectories
2: Output: List<List<double[]» selectedTrajectories
3:
4: def InitializeTrajectoryMinimumEnergy(trj):
5: selectedTrajectories = []
6: for i = 0 to maxTrajectories do
7:     currentTrajectory = [allConfigurations[0][i]]
8:     currentPoint = allConfigurations[0][i]
9:     for j = 1 to allConfigurations.Count do
10:        currentConfig = allConfigurations[j]
11:        minEnergy = MAX
12:        minIndex = 0
13:        for k = 0 to currentConfig.Count do
14:            energy = CalculateWeight(currentPoint, currentConfig[k])
15:            if energy < minEnergy then
16:                minEnergy = energy
17:                minIndex = k
18:            end if
19:        end for
20:        currentPoint = allConfigurations[j][minIndex]
21:        currentTrajectory.Append(currentPoint)
22:    end for
23:    selectedTrajectories.Append(currentTrajectory)
24: end for
25: return selectedTrajectories
```

5 Results

This chapter shows the results obtained by following the approaches described in Chapter 4.
The implementation was carried out in `C#` NetFramework 4.6.2 using Visual Studio 2022 as the Integrated Development Environment (IDE).

5.1 Robot KUKA KR50 R2100

The proposed solution is independent of the choice of robot used. For the tests the KUKA KR50 R2100 Robot (Figure 5.1), a high precision and versatile industrial robot, is chosen. It is widely used in automation applications due to its robustness and ability to operate in complex environments.
Technical specifications:

- Model: KUKA KR50 R2100
- Capacity: 50 kg
- Range of action: 2100 mm
- Number of axies: 6

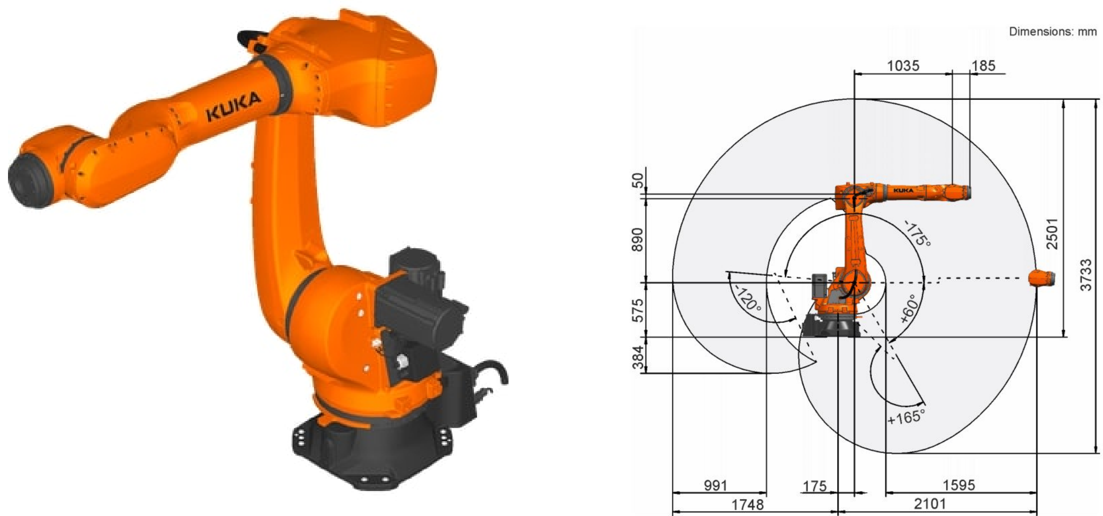


Figure 5.1: On the left: The KUKA KR50 R2100 robot displayed within its simulated working environment. On the right: The graphical representation of the robot's workspace.

5.2 The User Interface

For this thesis, a test program has been developed using the EuclidLabs 3D world as a test environment. As shown in Figure 5.2, the interface displays the robot and allows the user to import real-world trajectories or generate synthetic ones. Once the trajectory is imported, these trajectories are displayed as in Figure 5.3, with warning signals marking unreachable trajectory points.

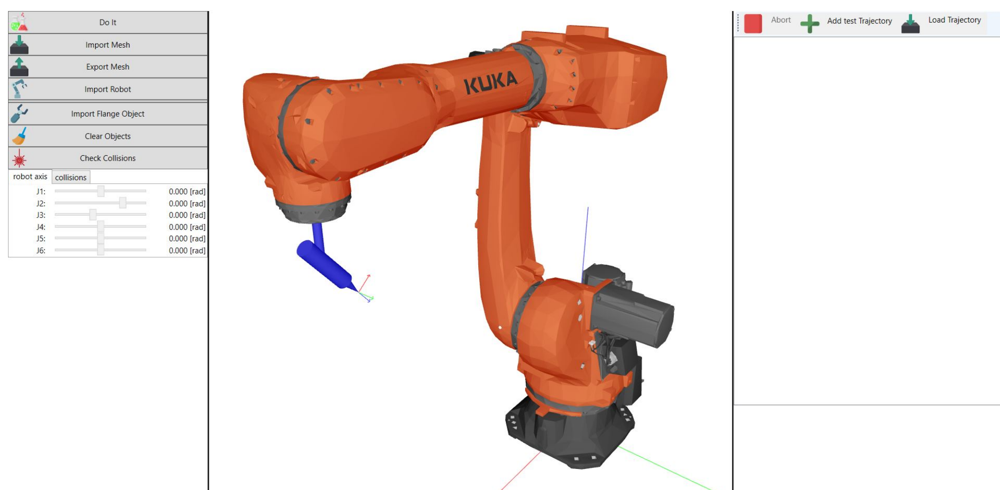


Figure 5.2: User Interface

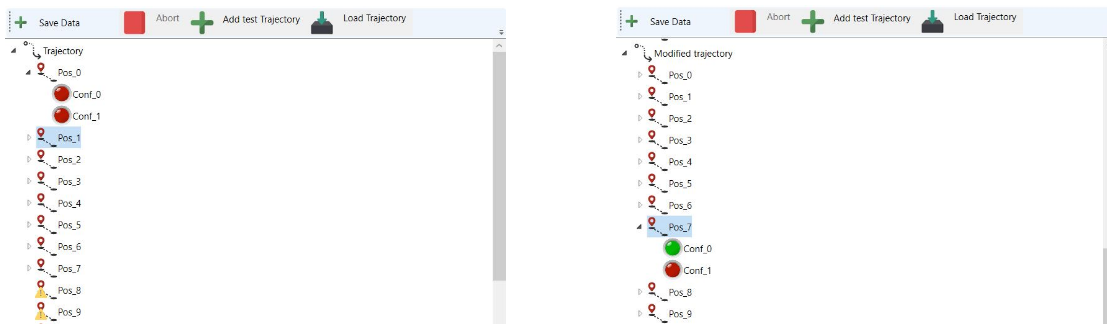


Figure 5.3: User interface: on the left, the imported trajectory is displayed with all trajectory points and configurations; on the right, the optimized trajectory is shown with selected configurations highlighted in green.

5.3 Synthetic Trajectories

Three different trajectories are used to test the proposed algorithms. The first one is a conical spiral, calculated as explained in Section 4.1.1 and shown in Figure 5.4. The second (Figure 5.5) is a spherical spiral option, implemented according to the method described in Section 4.1.2. The third is also a spiral trajectory, implemented using Equation 4.2 with $z(\phi) = 700$ ((Figure 5.6).

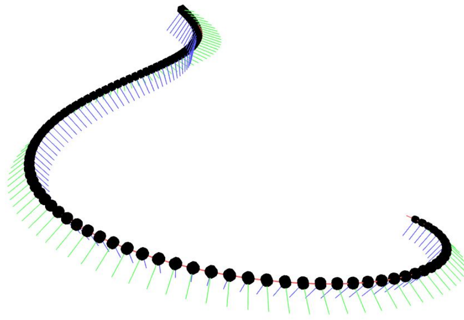


Figure 5.4: Conical spiral trajectory (100 samples) with $z_{\max} = 500$ mm, $a = \frac{z_{\max}}{m \cdot \text{maxRotation}}$, and $m = \cos\left(\frac{\pi}{6}\right)$.

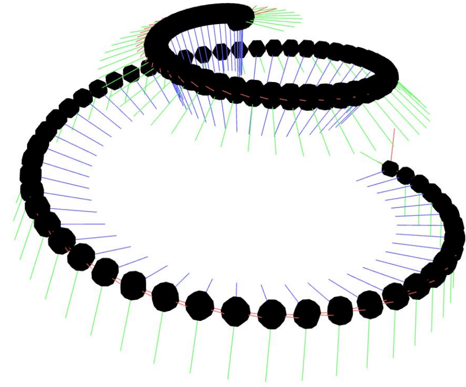


Figure 5.5: Spherical spiral trajectory (100 samples) with $r = 200$ mm and ϕ limited to $\frac{\pi}{2}$.

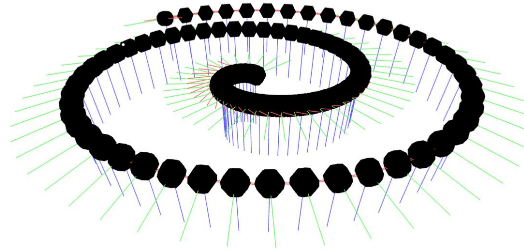


Figure 5.6: Planar spiral trajectory (100 samples) with $r = 200$ mm, ϕ limited to $\frac{\pi}{2}$ and $z(\phi) = 700$mm.

5.4 Real-World Trajectories

Real-world trajectories are recorded using MARVIN, a software provided by Euclid Labs that supports various industrial robots and enables hardware-independent trajectory acquisition. It allows users to manually record complex trajectories executed by an operator and convert them into robot programs.
For this analysis, two real-world trajectories are utilized: one representing the application of glue to shoes (Figure 5.7, 258 poses) and the other involving the painting process of a bicycle (Figure 5.8, 188 poses). These examples are chosen to assess the performance of the proposed algorithms under practical conditions with different levels of complexity.

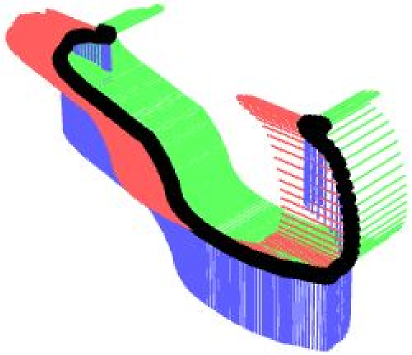


Figure 5.7: glue application to shoes.

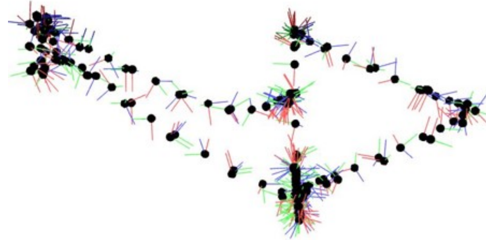


Figure 5.8: Bike painting process.

5.5 Results

The following sections present the results for the synthetic trajectory analysis, detailing the performance of the Greedy Graph Connectivity-Based Approach (Section 5.5.1), the Graph Energy Minimization Approach (Section 5.5.2), and a comparative analysis between the two Approaches (Section 5.5.3).
Section 5.5.4 extends the analysis to real-world trajectories, showcasing how both approaches perform under practical conditions.
Data collected include computation time, total energy, and the average/variance energy components. Specifically, for synthetic trajectories, these data are collected starting from 25 poses and increasing incrementally by 25, up to a maximum of 1000 poses.
It is important to note that, since all processing is conducted offline, we are not focused on achieving real-time performance, allowing for longer processing durations if they lead to improvements in trajectory efficiency and effectiveness.

5.5.1 Greedy Graph Connectivity-Based Approach

In some cases, the algorithm may encounter a local minimum, where it becomes unable to resolve the current level, leading to an excessively long recursion as the algorithm repeatedly attempts to fix the same level without success. To handle this, the `CreateGraph` function is modified to keep track of how many times it has attempted to fix the same level. If this happens more than 5 times, it restarts from the first level, randomizing all the poses. If the variable

maxRestart, which tracks the total restarts, exceeds 50, it concludes that no solution could be found within the allotted time. This behavior is indicated in the subsequent graphs by a red cross, showing that the algorithm was unable to find a valid solution.
Figure 5.9 illustrates the behavior of the variable countRestart, which tracks the number of times the algorithm restarts in a conical trajectory. This data highlights that the algorithm often gets stuck in local minima, requiring repeated restarts to overcome these challenges and continue the search for a feasible path.

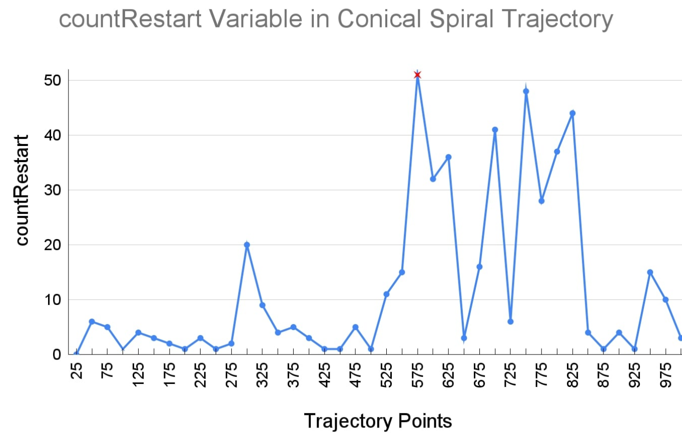


Figure 5.9: The countRestart variable in conical spiral trajectories.

Figure 5.10 shows the absolute value of the difference in energy between two test runs ($|E_1 - E_2|$). The variability between tests indicates that the algorithm does not consistently find the minimum energy solution, demonstrating the process's stochastic nature and limitations in achieving optimal results.

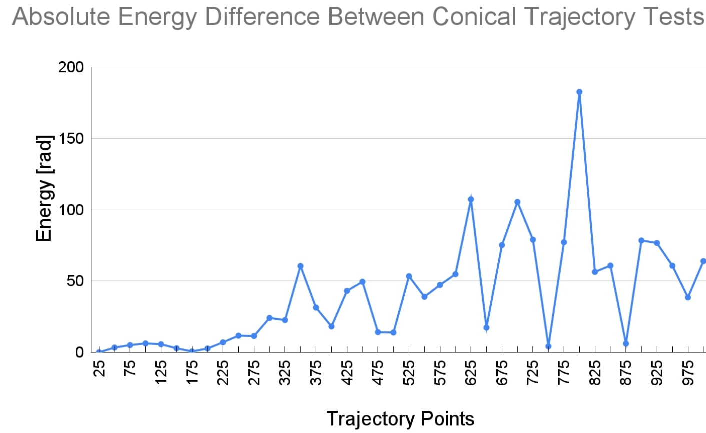


Figure 5.10: Absolute difference in energy between two different tests ($|E_1 - E_2|$).

5.5.2 Graph Energy Minimization Approach

The variable `maxTrajectories` represents the number of feasible configurations for the initial trajectory point, and its value directly affects the number of generated trajectories that are subsequently evaluated and compared. Figure 5.11 shows the comparison of energy outcomes when setting maxTrajectories to different values. The results highlight a noticeable difference between `maxTrajectories` = 30 and `maxTrajectories` = 5, showing that with only 5 configurations the algorithm is limited in achieving minimum energy values.

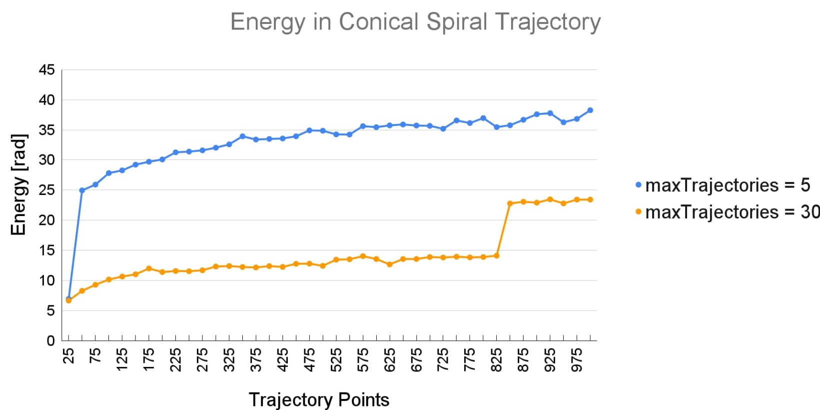


Figure 5.11: Energy results for different `maxTrajectories` values.

5.5.3 Comparison of Approaches for Synthetic Trajectories

Figures 5.12, 5.13, and 5.14 illustrate a comparison between the two approaches for synthetic trajectories in terms of time, energy, mean and variance. The results shown are calculated as the average of two test runs.
In the Greedy approach, we set an acceptable weight of 1 between configurations. For the Graph Energy Minimization Approach, the variable `maxTrajectories` is set to 30 to allow a more extensive exploration of feasible configurations.

For simpler trajectories, such as the planar one, the Greedy approach finds solutions faster than the other method but with higher energy values, as it stops at the first acceptable solution without attempting to further minimize the energy cost. However, for more complex trajectories, such as the conical and spherical ones, the Greedy approach faces significant challenges. The frequent orientation changes in these trajectories make the method unreliable, resulting in frequent failures to find a solution. Table 5.1 highlights the specific trajectory points where the Greedy approach stopped during the tests, underscoring its limitations in handling complex paths and the frequency with which it gets stuck in local minima. On the other hand, the Graph Energy Minimization approach consistently finds lower and more stable energy values, even if it requires longer computational time due to its exploration of feasible configurations.

Trajectory	Test	Trajectory Points	Stopped Level
Cone	1	800	711
Cone	2	1000	716
Cone	2	575	517
Cone	2	900	738
Sphere	1	25	19
Sphere	1	575	535
Sphere	1	950	897
Sphere	1	975	765
Sphere	2	25	19
Sphere	2	825	794
Sphere	2	875	740

Table 5.1: Stopped levels for the conical trajectory

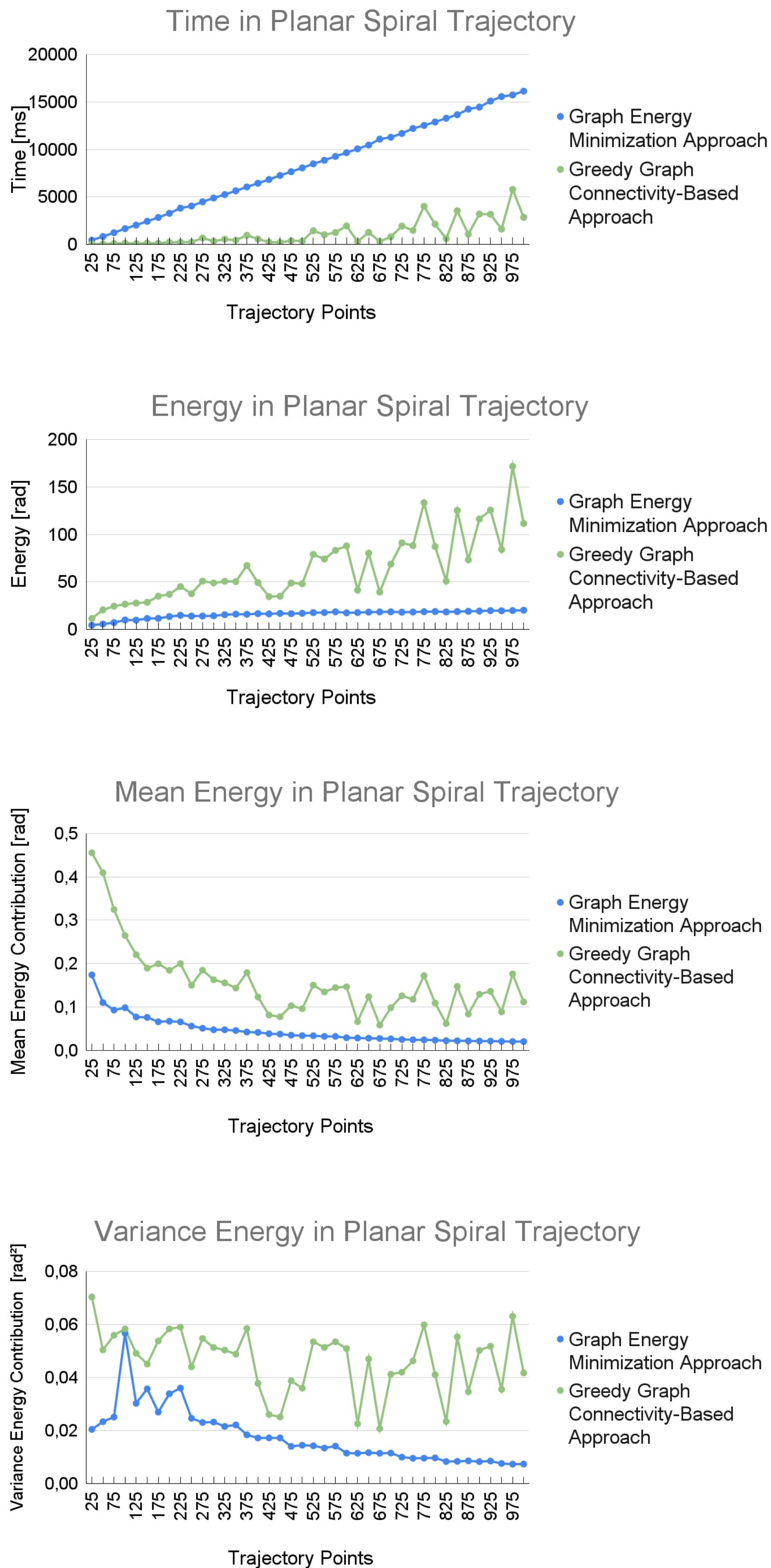


Figure 5.12: Results for the planar spiral trajectory.

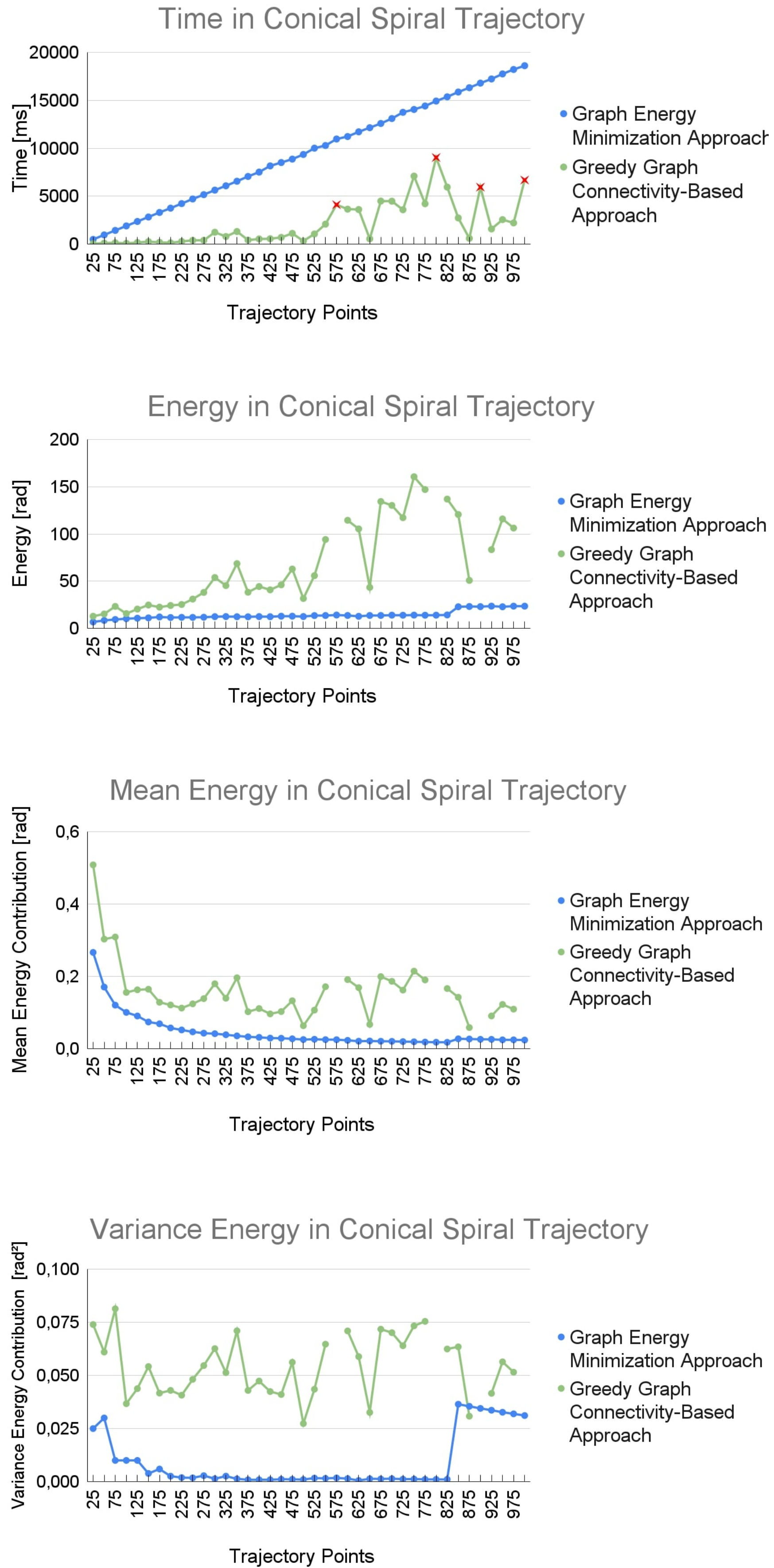


Figure 5.13: Results for the conical spiral trajectory.

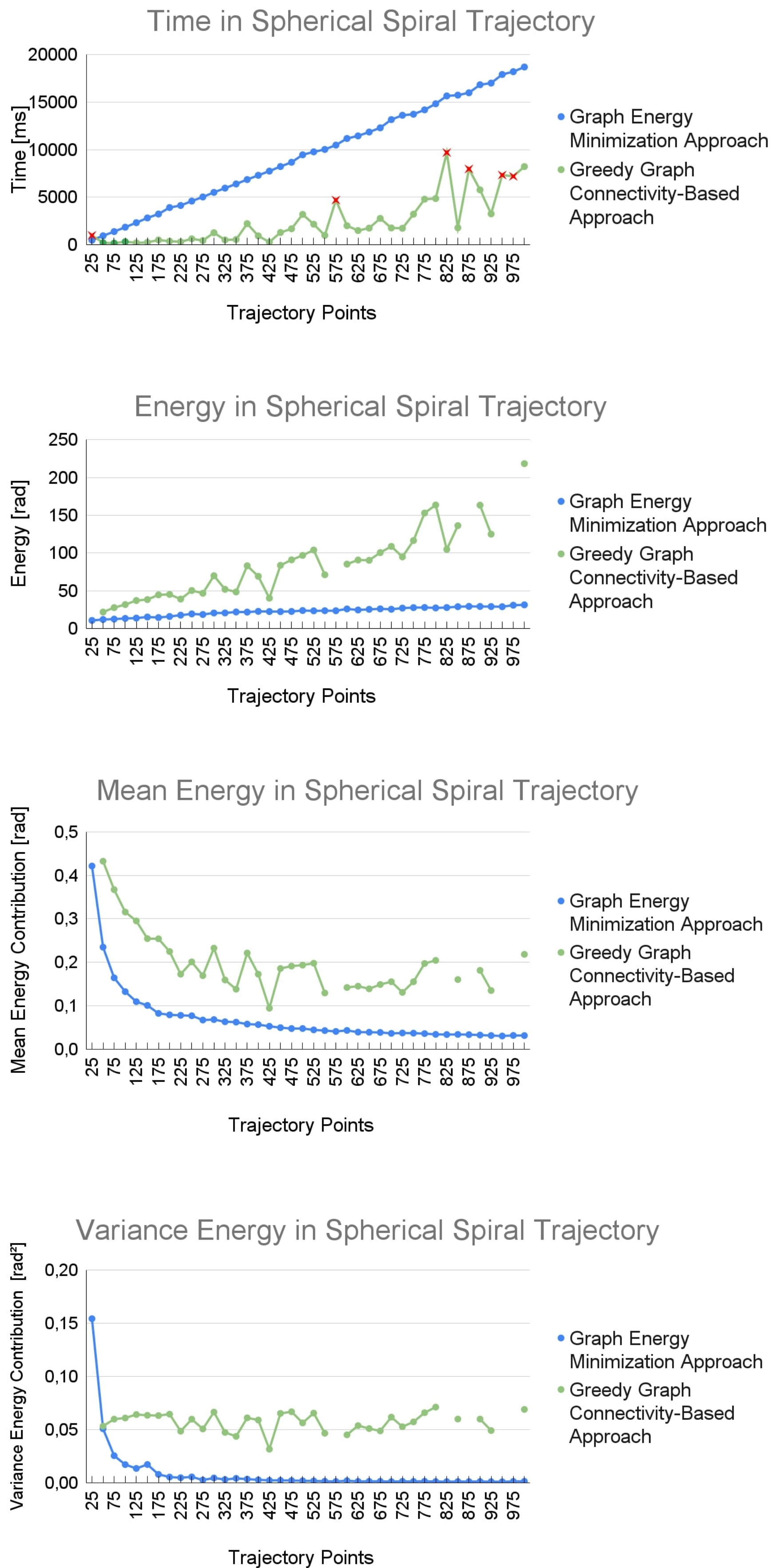


Figure 5.14: Results for the spherical spiral trajectory.

5.5.4 Comparison of Approaches on Real-World Trajectories

The tests conducted on real-world trajectories confirm the observations made during the synthetic trajectory experiments. Specifically, for the glue application to shoes, which involves a trajectory with few orientation changes, the Greedy Graph Connectivity-Based Approach is faster and finds an energy value comparable to that of the other approach. However, since the bicycle painting trajectory requires handling multiple orientation changes, the Greedy approach fails to find a solution as it often gets stuck in local minima. In contrast, the Graph Energy Minimization Approach successfully finds a solution.
The collected data are shown in Table 5.2.

Trajectory	Approach	Time (ms)	Energy [rad]	Average Energy Component [rad]	Variance [rad^2]
Glue application	Greedy	64	0.7345	0.0028	2.0588E-06
Glue application	Energy	5792	0.6675	0.0026	1.5980E-06
Bicycle Painting	Energy	4518	240.3861	1.2786	0.7189

Table 5.2: Comparison of computation time, total energy, average energy component, and variance for real-world trajectories.

Conclusions and Future Works

In recent years, the development of algorithms for robotic optimization in the industrial sector has become increasingly necessary. For this thesis the problem to determine a feasible trajectory that allows the robot to move smoothly along a desired path has been faced. The primary focus was on minimizing the energy associated with joint movements, defined as the sum of the changes of the joints along the trajectory. To achieve this, the rotation around the Tool Axis was used as a degree of freedom to create efficient trajectories that overcome reachability issues, avoid singularities, and optimize the robot's movements.
To this aim, two methods based on graph construction were implemented. The first one is a greedy approach that examines each point in sequence, rotating it until an acceptable solution is found. This method often fails because it tends to get stuck in local minima, limiting its effectiveness for more complex trajectories. To overcome this limitation, a second algorithm was implemented to explore the joint space more systematically, aiming to find not only valid but also optimal solutions. The Graph Energy Minimization approach demonstrated a greater ability to produce consistent trajectories, minimizing joint motion and improving adaptability to complex trajectories, such as spherical and conical ones.
Both methods were tested on real trajectories generated by the MARVIN software, confirming the results observed with the synthetic ones. In the tests, the application of glue produced similar results between the two approaches, as this type of trajectory involves few changes in direction and can be managed effectively by the greedy method. However, in the painting of bicycles the Graph Energy Minimization approach proved significantly superior, validating that the

greedy approach is reliable only for less demanding trajectories.

6.1 Future Works

As possible future improvements, we propose the following:

- Addressing local minima in energy optimization problems
  One of the main challenges in finding suitable algorithms for this problem has been the non-linear nature of the energy landscape, which features many local minima. This makes it difficult to identify in advance which points are worth exploring to minimize energy without merely falling into local minima. One potential solution is to use stochastic optimization methods, such as simulated annealing or evolutionary algorithms, which introduce randomness to escape local minima and explore a broader range of the energy landscape.

- Variable-step sampling refinement
  A possible improvement could involve refining the sampling process. Currently, the algorithm performs a basic sampling of the axis space, generating configurations by rotating by 1 degree around the Z-axis. Since minimizing execution time is a secondary goal, an additional refinement phase could be introduced: once an acceptable trajectory is found, further minimization could be achieved by using variable-step sampling. For instance, applying a bisection method between configurations would allow for a more accurate exploration of configurations.

- Increased degrees of freedom
  Effective energy minimization could be improved by increasing the degrees of freedom, allowing small rotations around other axes depending on the specific application. This approach could provide greater flexibility and precision, especially in cases where additional degrees of freedom are feasible. For example, the STAMPEDE method demonstrates improved results in trajectory optimization by using rotations around all axes.

- Integration of real-world constraints
  In practical applications, additional constraints may need to be considered:

- Certain configurations might be preferred to avoid damaging components attached to the robot, potentially limiting the allowable range of rotation on some axes. In these cases, virtual axis limits that differ from the robot's physical limits could be introduced, ensuring compatibility with the current implementation.
- To account for potential environmental collisions, a cost function could be designed to not only minimize the motion of the robot's axes but also to maximize the safety distance from surrounding objects. This approach would enhance the robot's adaptability to complex environments by balancing efficient motion with safety and operational constraints.

References

[1] Andreas Aristidou and Joan Lasenby. "FABRIK: A fast, iterative solver for the Inverse Kinematics problem". In: *Graphical Models* 73.5 (2011), pp. 243–260.

[2] Jon Stojan. *The future of AI in robotics: industry perspectives and technological advancements*. 2023. URL: https://www.digitaljournal.com/tech-science/the-future-of-ai-in-robotics-industry-perspectives-and-technological-advancements/article.

[3] Euclid Labs. *Euclid Labs Official Website*. https://www.euclidlabs.it/. 2024.

[4] Daniel Rakita, Bilge Mutlu, and Michael Gleicher. "Stampede: A discrete-optimization method for solving pathwise-inverse kinematics". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 3507–3513.

[5] Phillipe Cardoso Santos et al. "M-FABRIK: A new inverse kinematics approach to mobile manipulator robots based on FABRIK". In: *IEEE Access* 8 (2020), pp. 208836–208849.

[6] Lorenzo Sciavicco and Bruno Siciliano. *Modelling and control of robot manipulators*. Springer Science & Business Media, 2012.

[7] StartUs Insights. *Top 10 Robotics Trends and Innovations in 2025*. 2024. URL: https://www.startus-insights.com/innovators-guide/robotics-trends-innovation/.

[8] Wikipedia contributors. *Depth-first search — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Depth-first_search.

[9] Mohd Zubair et al. "Kinematic mapping of Exoskeleton with virtual KUKA robot". In: *2016 International Conference on Robotics and Automation for Humanitarian Applications (RAHA)*. IEEE. 2016, pp. 1–5.

Acknowledgments

Vorrei ringraziare, nella mia lingua natia, tutti coloro che mi hanno sostenuto lungo questo percorso.

Desidero innanzitutto ringraziare Roberto Polesel per avermi dato la possibilità di svolgere questa attività all'interno dell'azienda Euclid Labs s.r.l. e il mio tutor aziendale Paolo Scremin che mi ha seguita con disponibilità e dedizione. Grazie per avermi fornito spunti fondamentali nella stesura di questo lavoro e per avermi indirizzata nei momenti di indecisione.
Ringrazio inoltre il Professor Ruggero Carli per aver accettato l'incarico di relatore per la mia tesi.

Dedico questo traguardo alla mia famiglia , in particolare mio papà Roberto e mia mamma Flora. Vi ringrazio per tutto quello che mi avete insegnato e messo a disposizione permettendomi di arrivare fin qui oggi.

È stato un percorso sfidante, non facile in cui si sono alternati periodi di motivazione a altri di sconforto. Per questo sento di ringraziare di cuore i miei amici per essermi sempre stati vicino e per avermi supportato nei momenti difficili. Grazie, in particolare, a chi c'è sempre e non smette di credere in me.

Voglio dedicare uno spazio alle persone che ho conosciuto durante questo percorso universitario. Padova e Aveiro mi hanno accolta lentamente, dimostrando che "casa" non è sempre e solo il luogo in cui si nasce , ma può essere una persona, un'emozione, o un insieme di persone e emozioni. Queste città mi hanno inaspettatamente donato tante "case" in cui ridere, riflettere, condividere, che poi sono diventate parte della mia quotidianità.
Amici miei, grazie per la leggerezza con cui mi avete accompagnata sopportando

lacrime e crisi esistenziali.

Durante questi 5 anni ho imparato che un'esperienza non vissuta è sicuramente un'occasione persa, ma anche che le opportunità non finiscono mai davvero. Ogni volta che ci si ferma c'è sempre un modo per riprovare con la consapevolezza di poter fallire ma anche quella che un modo per rimediare si trova, sempre.
Mi auguro, quindi, di sbagliare tante volte ma di avere altrettante opportunità, preparata al peggio ma pronta al meglio per questo cammino lavorativo che "inizia ora e finisce quando deve finire".

Appendix

A.1 Denavit-Hartenberg convention

The *Denavit-Hartenberg convention* (DH) is a standard method used in robotics to represent robot geometries; in particular, to describe the positions and orientations of segments of robotic manipulators with respect to a coordinate system. This method standardizes the process of deriving transformation matrices that describe the position and orientation of each joint and segment of the robot.
Consider Figure A.1; let Axis i denote the axis of the joint connecting Link $i-1$ to Link i; the so-called *Denavit-Hartenberg convention* (DH) can be written in an operating form as follows.

1. Find and number consecutively the joint axes (for a revolute joint, the axis about which the rotation is performed; for a prismatic joint, the axis along which the translation is performed); set the directions of axes $z_0, \ldots, z_{n-1}$.

2. Choose Frame 0 by locating the origin on axis z_0; axes x_0 and y_0 are chosen so as to obtain a right-handed frame. If feasible, it is worth choosing Frame 0 to coincide with the base frame.

Execute steps from 3 to 5 for $i = 1, \ldots, n-1$:

4. Locate the origin O_i at the intersection of z_i with the common normal to axes z_{i-1} and z_i. If axes z_{i-1} and z_i are parallel and Joint i is revolute,

then locate O_i so that $d_i = 0$; if Joint i is prismatic, locate O_i at a reference position for the joint range, e.g., a mechanical limit.

5. Choose axis x_i along the common normal to axes z_{i-1} and z_i with direction from Joint i to Joint $i + 1$.

6. Choose axis y_i so as to obtain a right-handed frame.

To complete:

7. Choose Frame n: if Joint n is revolute, then align z_n with z_{n-1}; otherwise, if Joint n is prismatic, then choose z_n arbitrarily. Axis x_n is set according to step 4.

8. For $i = 1, \ldots, n$, form the table of parameters $a_i, d_i, \alpha_i, \theta_i$:

   - a_i: distance between O_i and O_{i-1},
   - d_i: coordinate of O_{i-1} along z_{i-1},
   - α_i: angle between axes z_{i-1} and z_i about axis x_i to be taken positive when rotation is made counter-clockwise,
   - θ_i: angle between axes x_{i-1} and x_i about axis z_{i-1} to be taken positive when rotation is made counter-clockwise.

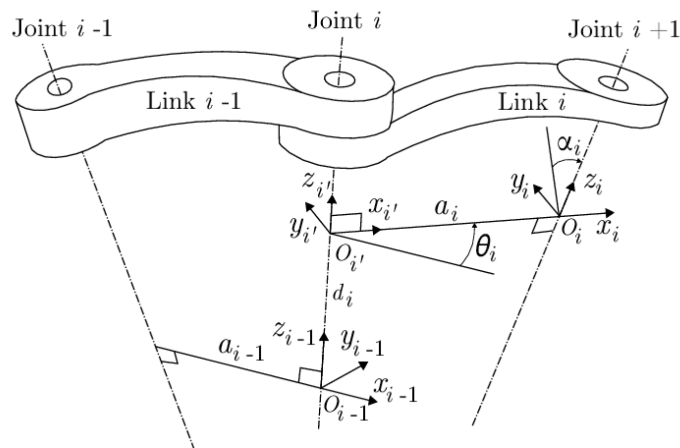


Figure A.1: Denavit-Hartenberg kinematic parameters