# POLITECNICO DI TORINO

**Master's Degree in Mechatronic Engineering**


# Inverse Kinematics Optimization for Robotic Manipulators: Exploiting Process-Based Redundancy

**Internal Supervisor**
**Prof. Marina Indri**

**Candidate**
**Alice Prunotto**

**External Supervisors**
**Ing. Antonio Venezia**
**Ing. Simone Panicucci**

**April 2025**

# Abstract

In industrial and collaborative environments, robotic manipulators are extensively used due to their precision and adaptability in performing complex tasks. Inverse kinematics is used to determine the joint configurations needed to achieve a target end-effector pose, which is given by both position and orientation. In redundant systems, where the number of degrees of freedom exceeds the minimum required for reaching the target, multiple inverse kinematics solutions exist. This redundancy admits additional optimization objectives, used to enhance the robot's performance.

The aim of the thesis project is to develop an IK algorithm that allows a robotic arm to reach a specified target, while optimizing additional factors. The end-effector goal configuration is composed by a full defined target position $(x, y, z)$ and a partial defined target orientation: the angles on the $x$ and $y$ axes are fixed, while the angle on the $z$ axis is left unconstrained. Rotations are usually treated with quaternions that represent a full orientation, making it difficult to isolate individual axes. To address this, the BioIK package from MoveIt is studied and used, and a customized goal class is developed, enforcing partial orientation constraints fixing two axes, but leaving rotation on the third axis unconstrained. This introduces a level of redundancy that allows the optimization of additional factors, such as avoiding joint limits and minimizing joint displacement. In addition to these goals, the final configuration must be collision-free.

The proposed approach has been tested both in a simulated environment using ROS and MoveIt!, and on a real Comau Racer-5 COBOT, to demonstrate its capability to achieve optimized collision-free inverse kinematics solutions. The results confirm its effectiveness in reaching the final target while optimizing movement, respecting kinematic constraints, and ensuring safe and feasible configurations.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background

In recent years robotics has become an essential part of modern industry, automation and research, changing the way in which several activities are executed in different areas. From industrial and logistic production, health and space exploration, robots have a crucial role in making efficiency, precision and security grow. These systems can operate in an autonomous way or collaborate with humans, executing both very easy and repetitive or more difficult and high computational tasks.

An important field within robotics is that of robotic manipulators. They are used both in industrial and collaborative environments thanks to their accuracy, precision and safety. These robotic arms perform several usages, such as assembly, welding and assisting in surgery, achieving precise movement control, interacting with objects and obstacles in the environment, and accurately reaching target positions and orientations.

Regardless of the purpose, a fundamental aspect in robotics is to solve inverse kinematics (IK), which allows to determine the joint configurations needed to reach an end effector target pose. The target pose is composed of a position $(x, y, z)$ and an orientation (that can be represented in different ways, such as quaternions or Euler angles). If the robot has few joints, the inverse kinematics is trivial, while if the kinematic chain is more complex, the inverse kinematics becomes more difficult. And it gets even more challenging in the case of redundant systems.

Redundant robots are manipulators that have more degrees of freedom than those strictly necessary to reach a target goal. As a consequence to this, an infinite number of solutions is possible, which allows to optimize other criteria as the avoidance of singularities, energy reduction, minimization of joint movements, and ensuring the final configuration remains collision-free. This characteristic can be useful for improving the system's performance, and an efficient optimization algorithm needs to be used to exploit redundancy effectively.

## 1.2 Problem statement

The thesis work was carried out in collaboration with Comau, a global industrial automation and robotics leader. It provided the necessary hardware and resources to apply and verify the introduced inverse kinematics algorithm on the Racer5-COBOT industrial robot manipulator.
The aim of the thesis project is to develop an inverse kinematics algorithm that allows a robotic arm to reach a specified target, while optimizing additional factors. The end-effector goal configuration is composed of a full defined target position $(x, y, z)$ and a partial defined target orientation: the rotation angles about the $x$ and $y$ axes are fixed, while the angle about the $z$ axis is left unconstrained. This particular goal configuration introduces redundancy, which is used to improve the manipulator performance by optimizing secondary objectives. The final configuration must be collision free.
One of the challenges lies in handling orientation constraints. In traditional IK solvers, rotations are expressed using quaternions, which provide a mathematical notation for representing spatial orientations and rotations of elements in three dimensional space. However, they define a complete rotation, making it impossible to impose constraints on individual axes.
To overtake this limitation and handle redundancy efficiently, an advanced IK solver capable of optimizing goals and handling flexible constraints is required.

## 1.3 Proposed approach

The BioIK package for MoveIt was studied and used to reach the objective. MoveIt is a flexible and powerful motion planning framework for ROS-based robotic systems. BioIK is a memetic inverse kinematics solver developed for the motion planning framework MoveIt and the robot operating system ROS. The memetic algorithm uses a combination of evolutionary optimization, particle swarm optimization, and gradient based methods. It defines several goal classes, each of which handles a specific objective. Goals can be primary or secondary, each being weighted and combined to create balance between the targets. BioIK enables users to personalize both goals and constraints.
To overcome the limitations of quaternion-based constraints, a customized goal class *PartialOrientationGoal* was implemented. It works with Roll, Pitch and Yaw (RPY) angles, bypassing quaternions. Additionally, secondary goals were included, to enhance the robot performance.

The algorithm was implemented in ROS1 using the MoveIt! framework, and programmed in C++, and the used robot is the Racer5-COBOT by Comau, equipped with a asymmetric rectangular end effector. The proposed approach was verified both with a simulation test and with a real environment test.
The performance of the algorithm is analyzed based on several metrics, including accu-

racy, to measure the precision with which the target goal is reached, computation time, optimization effectiveness on secondary goals, and collision avoidance.

## 1.4 Thesis structure

The thesis has the following structure:

- Chapter 2: an overview of robotic manipulators and basic robotic knowledge, as kinematics, inverse kinematics, redundancy and rotation representation is presented.
- Chapter 3: the principles of optimization are presented, among with three genetically inspired algorithms: evolutionary algorithms, particle swarm optimization and memetic algorithms.
- Chapter 4: the main characteristics of Robot Operating System (ROS) and the MoveIt motion planning framework are presented.
- Chapter 5: Racer5-COBOT, the robotic manipulator used for simulations is presented.
- Chapter 6: the BioIK package and kinematic solver is analyzed and studied.
- Chapter 7: the task, the simulation setup, environment and constraints are presented.
- Chapter 8: the experimental results are discussed.
- Chapter 9: presents the conclusions of the thesis and possible future work and developments that could improve the proposed solution.

# Chapter 2

# Robotics background

Robotics is a field that combines mechanics, electronics, computer science and engineering to build, design and interact with robots. Robots are autonomous or semi-autonomous machines capable of sensing their environment, carrying out computations to make decisions, and performing complex actions in the real world. The goal is to make efficiency, precision and security grow. These systems can operate in an autonomous way or collaborate with humans, executing both very easy and repetitive or more difficult and high computational tasks. They find applications in several fields, such as industry, agriculture, space, medicine and more.
Robotic manipulators are used both in industrial and collaborative environments thanks to their accuracy, precision and safety. These robotic arms perform several usages, such as assembly, welding and assisting in surgery, achieving precise movement control, interacting with objects and obstacles in the environment, and accurately reaching target positions and orientations. A manipulator is made of an arm, the first three links, and a wrist, the last links [2].

## 2.1 Kinematics

Kinematics deals with the study of the movement of the robotic arm with respect to a reference frame, without considering the forces that cause that motion. It allows to represent positions, velocities and accelerations of specified points in a multi-body structure. Direct kinematics consists of determining the end effector pose (position and orientation), given the joint variable values. Inverse kinematics consists of finding the right joint configurations that allows to achieve a predefined end effector pose.

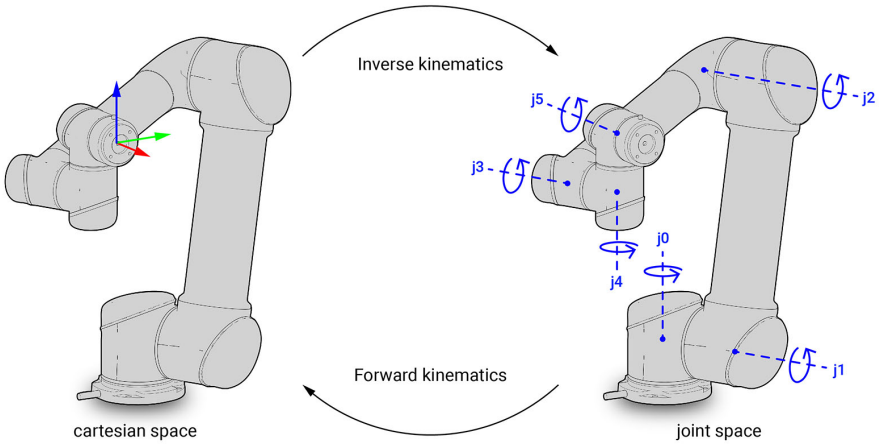


**Figure 2.1:** Forward and inverse kinematics [1]

### 2.1.1 Pose of a rigid body

The pose of a rigid body is defined by the position and the orientation with respect to a predefined reference frame. A reference frame is attached to the center of gravity of the rigid body, and its units vectors are expressed with respect to another reference frame.

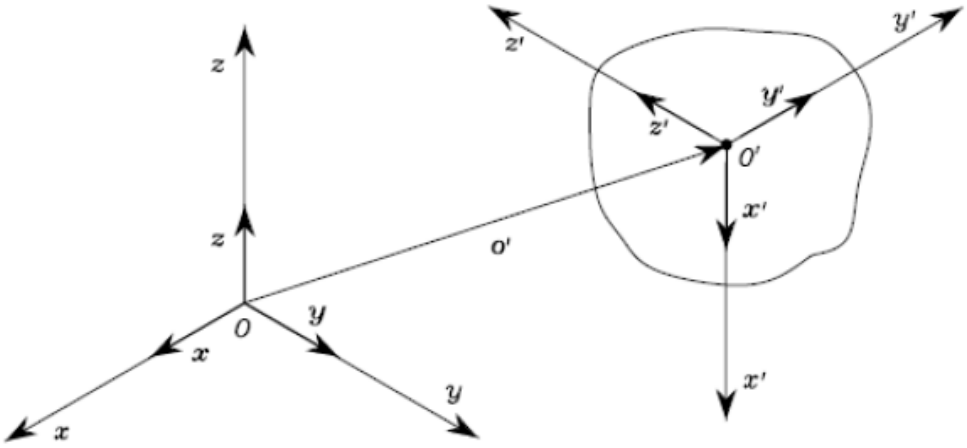


**Figure 2.2:** Pose of a rigid body [2]

Considering the fixed orthonormal reference frame $O - xyz$ with $x, y, z$ as axes, and the orthonormal body frame $O' - x'y'z'$ with $x', y', z'$ as axes [2], the position of point $o'$ with respect to the reference frame $O - xyz$ is:

$$o' = o'_x x + o'_y y + o'_z z \tag{2.1}$$

where $o'_x, o'_y, o'_z$ are the components of the vector $o'$ along the axes of the $O - xyz$ reference

frame. The position can be written as:

$$\mathbf{o}' = \mathbf{p} = \begin{bmatrix} o'_x \\ o'_y \\ o'_z \end{bmatrix} \tag{2.2}$$

The orientation, instead, is defined as:

$$x' = x'_x x + x'_y y + x'_z z \tag{2.3}$$

$$y' = y'_x x + y'_y y + y'_z z \tag{2.4}$$

$$z' = z'_x x + z'_y y + z'_z z \tag{2.5}$$

The orientation can be expressed by a rotation matrix $\mathbf{R}$:

$$\mathbf{R} = \begin{bmatrix} x'_x & y'_x & z'_x \\ x'_y & y'_y & z'_y \\ x'_z & y'_z & z'_z \end{bmatrix} \tag{2.6}$$

Combining the position and the orientation, the pose of the rigid body is obtained, and it can be summarized into a transformation matrix $\mathbf{T}$:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{p} \\ 0 & 1 \end{bmatrix} \tag{2.7}$$

where $\mathbf{R}$ is the rotation matrix and $\mathbf{p}$ is the position vector.
To transform the pose of a body from one reference frame to another, the transformation matrix is used.

### 2.1.2 Kinematic chains

A kinematic chain is an assembly of rigid bodies (links) connected by joints that defines the structure and movement of a robotic system. A reference frame is placed on each arm/link. There are two kinds of kinematic chain:

- Open kinematic chain: the last link is not connected to the system base.
- Closed kinematic chain: the last link is connected to the base or to another joint.

Kinematic chains are composed by links and joints.

#### Links

The links are the rigid members of a robot that are connected by joints and constitute the structure of the robot.

### Joints

Joints connect two consecutive links, allowing movements between them. Each joint allows one degree of motion, when a joint is not acuated it is called a passive joint. A scalar variable is associated to each joint, called joint variable $q$. There are two different kinds of joints:

- Prismatic: they allow a linear movement along one axis and the position of the joint is defined by a translation.
- Revolute: they allow a rotation about one axis and the configuration of the joint is defined by a rotation angle.

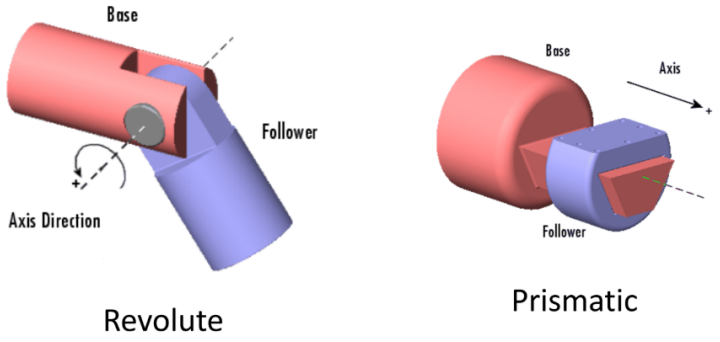


**Figure 2.3:** Joint types [3]

### 2.1.3 Workspace

The workspace of the robot is the region described by the origin of the end effector frame, considering all the possible motions of all the joints. It is the set of points in the space that the robot end effector can reach. In the data sheets it is reported as side view and top view.
Different types of workspace:

- Reachable workspace: it is the workspace that the end effector frame can describe with at least one orientation.
- Dexterous workspace: it is the workspace that the end effector frame can describe with more than one orientation.
- Redundant workspace: it is the workspace that the end effector can reach with multiple different joint configurations.

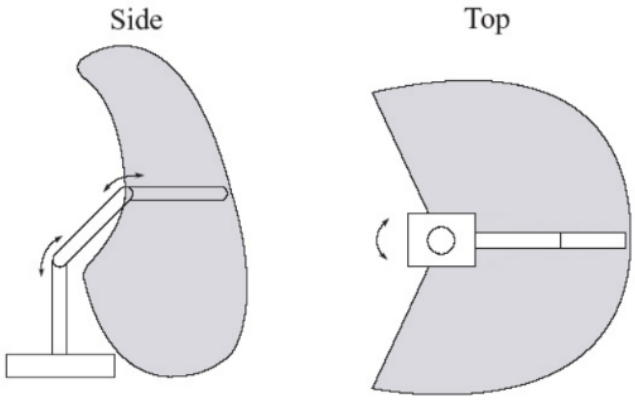


**Figure 2.4:** Workspace [4]

### 2.1.4 Redundancy

Redundancy happens when the system has more degrees of freedom than those strictly necessary to reach a target goal. This results in different configurations able to reach the same pose.
Redundant robots present several advantages, such as the ability to improve dexterity in confined spaces, avoid kinematic singularities, minimize energy and address many more secondary optimization objectives.
However, redundancy also causes challenges, primarily due to the fact that motion planning and control become increasingly complex. Since the additional degrees of freedom cause the inverse kinematics to have more than one solution, optimization algorithms are needed to choose the best one.

### 2.1.5 Denavit-Hartenberg convention

The Denavit-Hartenberg convention provides a way to describe the relationship between two consecutive links of a kinematic chain. It uses four parameters to describe the homogeneous transformation matrix that represents the relative position and orientation of one link with respect to the previous one (with reference to the Figure 2.5):

- $a_i$: link length, distance between origins $O_i$ and $O_{i'}$
- $\alpha_i$: link twist, angle between axes $z_{i-1}$ and $z_i$ about axis $x_i$
- $d_i$: link offset, coordinate of $O_{i'}$ along axis $z_{i-1}$
- $\theta_i$: joint angle, angle between axes $x_{i-1}$ and $x_i$ about axis $z_{i-1}$

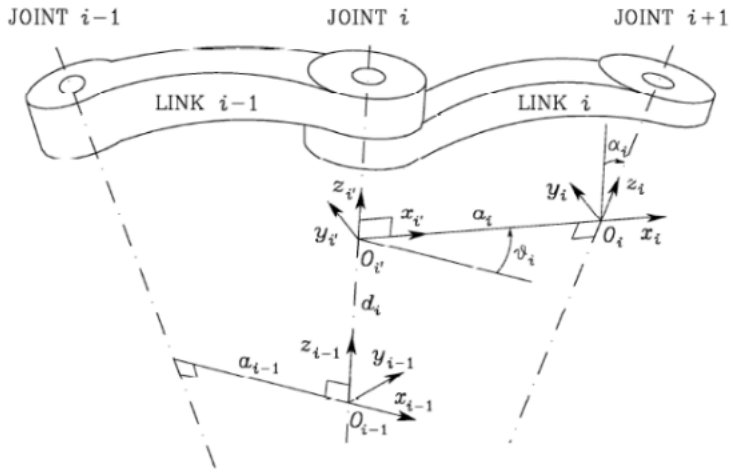


**Figure 2.5:** Denavit-Hartenberg parameters [4]

With these parameters, it is possible to write the homogeneous transformation matrix $\mathbf{T}_i^{i+1}$ that represents the transformation between two consecutive frames $i$ and $i+1$:

$$\mathbf{T}_i^{i+1} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

To obtain the overall transformation from the base frame to the end effector frame $\mathbf{T}_0^n$, the transformation matrices of the joints are multiplied:

$$\mathbf{T}_0^n = \mathbf{T}_1^2 \cdot \mathbf{T}_2^3 \cdot \cdots \cdot \mathbf{T}_{n-1}^n \tag{2.9}$$

### 2.1.6 Direct kinematics

Direct kinematics is the process of obtaining the pose and velocity of the end effector, knowing the joint values and the angular velocities, using kinematics equations.
Given a manipulator with $n$ joints and $n+1$ links, the joint variables are represented by a vector $q = [q_1, q2, \ldots, q_n]^T$, and the forward kinematics function is expressed as:

$$x = f(q) \tag{2.10}$$

where $x$ is the pose of the end effector and $f$ is the kinematic function for the robot, which depends on the kind of joints that are present.

### 2.1.7 Inverse kinematics

Inverse kinematics is the process used to determine the joint values needed to reach an end effector target position and orientation.

The inverse kinematics function is expressed as:

$$q = f^{-1}(x) \tag{2.11}$$

where $x$ is the pose of the end effector and $q$ is the vector containing the joint values. While direct kinematics is not overly complex and it has a unique solution, inverse kinematics is not so trivial and it presents some challenges:

- It is a non-linear problem, so closed-form solutions do not always exist.
- Solutions may not exist, since the pose target can be unreachable due to constraints or limitations. If the target pose belongs to the dexterous workspace of the robot, at least one solution is guaranteed.
- If the system is redundant, multiple solutions exist, and an optimization criteria is needed to choose the best one.

Examining the number of DOFs, $n$, of the robot and the number of the task constraints, $m$, it is possible to determine the number of solutions:

- If $n < m$, then no solutions exist, and the problem is called overconstrained.
- If $n = m$, then a finite number of solutions exist.
- If $n > m$, then either no solutions or an infinite number of solutions exist, the problem is called underconstrained and the robot is redundant.

Inverse kinematics can be solved in several ways, including analytical approach, numerical approach or using the Jacobian matrix.

### Analytical approach

When the system is not so complex, inverse kinematics can be found inverting analytically the forward kinematics equations, by deriving explicit equations for joint variables, often using geometric and trigonometric methods. The advantages are that it is possible to compute all IK solutions, and to determine whether no solution exists; once the equations are derived, the solutions are very fast to compute, and there is no need to define an initial guess or solution parameters. However, when the complexity of the robot increases, this method does no longer work.

### Numerical approach

Numerical approaches work by iteratively improve the IK solutions, until a sufficiently accurate one is found or the time limit is reached. An initial guess $q_0$ is given, and then a sequence of different configurations $q_1, q_2 \dots$ are calculated such that the error $\|f(q) - x\|$ approaches 0:

$$minimize \|f(q) - x\| \tag{2.12}$$

### Jacobian matrix

Various numerical approaches are based on the Jacobian matrix, $J$. It provides the mapping between joint velocities and the corresponding end-effector velocity.
In the operational space there are two types of velocity of the end effector: an analytical one, and a geometrical one.
The analytical velocity is defined as:

$$\dot{x} = \begin{bmatrix} \dot{\mathbf{p}} \\ \dot{\phi} \end{bmatrix} \tag{2.13}$$

where $\dot{\mathbf{p}}$ is the vector containing the linear velocities and $\dot{\phi}$ is the vector containing the rates of change of the orientation representation.
The analytical Jacobian matrix $J_a$ is the matrix that relates the joint velocities $\dot{q}$ to the analytical velocity $\dot{x}$:

$$\dot{x} = J_a \dot{q} \tag{2.14}$$

and it is defined as:

$$J_a(q) \equiv \frac{\partial f(q)}{\partial q} \equiv \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \cdots & \frac{\partial f_1}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial q_1} & \cdots & \frac{\partial f_m}{\partial q_n} \end{bmatrix} \tag{2.15}$$

The geometrical velocity is defined as:

$$\mathbf{v} = \begin{bmatrix} \dot{\mathbf{p}} \\ \omega \end{bmatrix} \tag{2.16}$$

where $\omega$ is the vector containing the angular velocities of the end effector.
The geometrical Jacobian matrix $J_g$ is the matrix that relates the joint velocities $\dot{q}$ to the geometrical velocity $\mathbf{v}$:

$$\mathbf{v} = J_g \dot{q} \tag{2.17}$$

It can be divided into two sub-matrices, one for linear velocity and one for angular velocity:

$$J_g = \begin{bmatrix} J_p \\ J_w \end{bmatrix} = \begin{bmatrix} J_{l,1} & J_{l,2} & \cdots & J_{l,n} \\ J_{a,1} & J_{a,2} & \cdots & J_{a,n} \end{bmatrix} \tag{2.18}$$

where $J_{l,i}$ defines the influence of the $i^{th}$ joint on the end effector linear velocity, and $J_{a,i}$ defines the influence of the $i^{th}$ joint on the end effector angular velocity.
The Jacobian matrix is used to solve inverse kinematics. The idea is to iteratively update the joint positions until the end effector reaches the desired pose. First, the difference between the target pose, $x_T$, and the current end effector pose, $x$, is calculated:

$$\Delta x = x_T - x \tag{2.19}$$

A small joint difference, $\Delta q$ is needed to reduce the pose difference. To find this value, it is necessary to invert the Jacobian:

$$\Delta q = J^{-1}\Delta x \tag{2.20}$$

If $J$ is square and invertible, the inverse can directly be calculated. If $J$ is not invertible, the Moore–Penrose inverse is used instead:

$$J^{+} = J^{T}(JJ^{T})^{-1} \tag{2.21}$$

Then, the small change in the joint velocities that brings the current position near to the end effector position is calculated:

$$\Delta q = J^{+}\Delta x \tag{2.22}$$

Finally, the values of the joints are iteratively updated until convergence is reached:

$$q_{n+1} = q_n + \alpha\Delta q \tag{2.23}$$

where $\alpha$ is a scale factor for stability.

### 2.1.8 Singularities

A singularity is a particular point in the workspace in which the robot looses one or more degrees of freedom (DoF), which is the number of independent movements a robotic arm can make. When the robot's tool center point (TCP, the ideal point on the end effector that the robot software moves through space) moves into or near a singularity, the robot will stop moving or move in an unexpected manner, making it difficult to control it. A robot gets into a singularity when the Jacobian matrix is singular, when it is not invertible or its determinant is zero.
Singularities are classified into three types [15]:

- Wrist singularity: it occurs when the axes of two joints line up exactly with each other. In robotic manipulators it happens when the axes of joint 4 and joint 6 become parallel and additionally they could also share a point.
- Elbow singularity: it occurs when the elbow joint of the robotic arm is fully extended or fully contracted, and the arm forms a straight line.
- Shoulder singularity: it occurs when the center of the robot's wrist aligns with the axis of the first joint, or when the axis of the last joint coincides with the axis of the first joint, making the shoulder joint align in a way that makes the arm lose some ability of movement.

## 2.2 Orientation representation

There are several orientation representations, that is the mathematical methods used to describe the rotation of an object in the space. A powerful representation must be brief, computationally effective, and singularity-free. Rotation matrices, quaternions, and Roll-Pitch-Yaw (RPY) angles are very popular representations, each with their strengths and weaknesses. The choice of representation will be application dependent, compromising between ease of interpretation, computational expense, and numerical stability.

### 2.2.1 Rotation matrix

The rotation matrix (already defined in Formula 2.6) represents the orientation of the body frame with respect to the reference frame. R is an orthogonal matrix, which means that it preserves the length and angles of vectors when applied, and the rows and the columns are mutually orthogonal and with unit norm. Hence, its transpose is equal to its inverse:

$$\mathbf{R}^T = \mathbf{R}^{-1} \tag{2.24}$$

$$\mathbf{R}^T\mathbf{R} = I_3 \tag{2.25}$$

If two reference frames have the same origin and one is rotated about a generic axis with respect to the other, it is possible to transform the coordinates of a point $\mathbf{p}$ in the reference frame $O - xyz$ into a point $\mathbf{p}'$ in the reference frame $O - x'y'z'$. The point is represented in the two reference frames as:

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}, \quad \mathbf{p}' = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} \tag{2.26}$$

The relations between the two points are defined as:

$$\mathbf{p} = \mathbf{R}\mathbf{p}' \tag{2.27}$$

and

$$\mathbf{p}' = \mathbf{R}^T\mathbf{p} \tag{2.28}$$

### 2.2.2 Quaternions

A quaternion is an orientation representation formed by four components, one scalar, and three vector components. A quaternion $q$ is usually written as $q = q_0 + q_1 i + q_2 j + q_3 k$ where $q_0$, $q_1$, $q_2$ and $q_3$ are real numbers. The scalar part is $q_0$, while the vector part is $(q_1, q_2, q_3)$ along the imaginary basis $i, j, k$, that satisfy the following rules:

- $i^2 = k^2 = j^2 = ijk = -1$
- $ij = k$, $ji = -k$, $jk = i$, $kj = -i$, $ki = j$, $ik = -j$

To ensure valid rotations, unit quaternions, which satisfy $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$, are used. A rotation by an angle $\theta$ around a unit axis $\mathbf{v} = (v_x, v_y, v_z)$ is converted into a quaternion with the following formula:

$$q = \cos\frac{\theta}{2} + (v_x i + v_y j + v_z k)\sin\frac{\theta}{2} \tag{2.29}$$

that can also be written as:

$$q = \left(\cos\frac{\theta}{2}, v_x \sin\frac{\theta}{2}, v_y \sin\frac{\theta}{2}, v_z \sin\frac{\theta}{2}\right) \tag{2.30}$$

Given a rotation matrix $\mathbf{R}$, the corresponding quaternion can be found as:

$$q_0 = \frac{1}{2}\sqrt{1 + r_{11} + r_{22} + r_{33}} \tag{2.31}$$

$$q_1 = \frac{1}{4}w(r_{32} - r_{23}), \quad q_2 = \frac{1}{4}w(r_{13} - r_{31}), \quad q_3 = \frac{1}{4}w(r_{21} - r_{12}) \tag{2.32}$$

Instead, a quaternion can be converted into a rotation matrix as follows:

$$\mathbf{R} = \begin{bmatrix} 1 - 2(q_2^2 + q_3^2) & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & 1 - 2(q_1^2 + q_3^2) & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & 1 - 2(q_1^2 + q_2^2) \end{bmatrix} \tag{2.33}$$

Quaternions are very useful in applications requiring smooth interpolation, such as robot motion planning. They provide a compact and efficient way to perform rotations using quaternion multiplication.

### 2.2.3 RPY angles

RPY angles describe a rotation that is composed of three sequential rotations about predefined axes:

- Roll: rotation by angle $\psi$ about fixed axis $x$
- Pitch: rotation by angle $\theta$ about fixed axis $y$
- Yaw: rotation by angle $\phi$ about fixed axis $z$

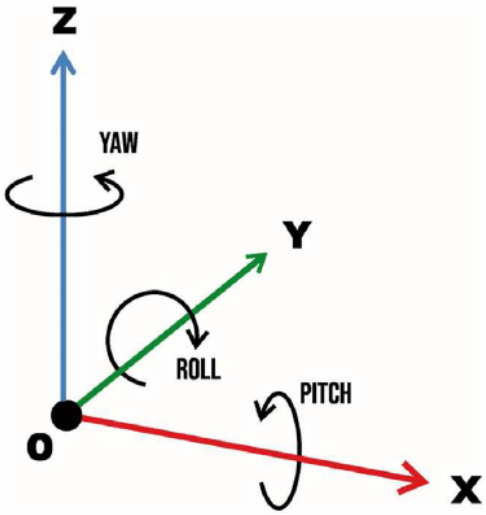


**Figure 2.6:** RPY angles [5]

The elemental rotation matrices are [16]:

$$\mathbf{R}_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\psi) & -sin(\psi) \\ 0 & sin(\psi) & cos(\psi) \end{bmatrix} \tag{2.34}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{2.35}$$

$$\mathbf{R}_z(\phi) = \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.36}$$

These three matrices can be combined into one matrix, by multiplying them in the correct order (here $cos(\alpha)$ is abbreviated to $c_\alpha$ and $sin(\alpha)$ is abbreviated to $s_\alpha$ for convenience and simplicity of notation):

$$\mathbf{R}(\Theta) = \mathbf{R}_z(\phi)\mathbf{R}_y(\theta)\mathbf{R}_x(\psi) = \begin{bmatrix} c_\phi c_\theta & c_\phi s_\theta s_\psi - s_\theta c_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ s_\phi c_\theta & s_\phi s_\theta s_\psi + c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi \\ -s_\theta & c_\theta s_\psi & c_\theta c_\psi \end{bmatrix} \tag{2.37}$$

Starting from a given a rotation matrix:

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.38}$$

it is possible to calculate corresponding RPY angles using the following formulas:

- Roll: $\psi = tan^{-1}(\frac{r_{21}}{r_{11}}) = atan2(r_{21}, r_{11})$
- Pitch: $\theta = -sin^{-1}(r_{31}) = -asin(r_{31})$
- Yaw: $\phi = tan^{-1}(\frac{r_{32}}{r_{33}}) = atan2(r_{32}, r_{33})$

To convert RPY angles into a quaternion $q = (w, x, y, z)$, the following formulas are used:

$$w = \cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \tag{2.39}$$

$$x = \sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} - \cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \tag{2.40}$$

$$y = \cos\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} \tag{2.41}$$

$$z = \cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} - \sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} \tag{2.42}$$

On the contrary, to convert a quaternion into RPY angles, $\psi$, the following formulas are used:

- Roll $\psi$:

$$\psi = \operatorname{atan2}\Big(2(wx + yz), 1 - 2(x^2 + y^2)\Big) \tag{2.43}$$

- Pitch $\theta$:

$$\theta = \operatorname{asin}\left(2(wy - zx)\right) \tag{2.44}$$

- Yaw $\phi$:

$$\phi = \operatorname{atan2}\Big(2(wz + xy), 1 - 2(y^2 + z^2)\Big) \tag{2.45}$$

# Chapter 3

# Optimization approaches

Optimization is widely used in several applications, like robotics, machine learning, computer science and more.
It is the process used to find the best solution to a given problem within the set of possible solutions, under certain constraints [17]. The goal of the optimization problem is to minimize (or maximize) a cost function, while guaranteeing that the constraints are satisfied.
There are different methods to solve an optimization problem:

- Numerical methods rely on mathematical formulations and systematic procedures to find optimal solutions. They are often accurate and deterministic, but they struggle with more complex or non-differentiable functions [18].
  - Gradient-based methods use derivatives to iteratively adjust variables in the direction of steepest descent, making them effective for optimizing smooth and differentiable functions.
  - Mathematical programming formulates an optimization problem with an objective function and some constraints. It can be solved using methods like linear programming, nonlinear programming, or integer programming.
- Heuristic methods do not use strict mathematical formulas, but instead intelligent search algorithms. They are not always accurate, but they do well with more complex and high-dimensional problems [19].
  - Evolutionary algorithms take inspiration from natural selection and use mechanisms like mutation, crossover, and selection to iteratively improve candidate solutions.
  - Swarm intelligence methods model the collective behavior of biological systems, like birds or ants, to search and optimize solutions.
  - Simulated annealing replicates the process of physical annealing, gradually reducing randomness to avoid local minima and find an optimal solution.

Since inverse kinematics usually involves more than one constraint, redundancy, and non-smooth objectives, heuristic solutions provide a good strategy to find acceptable

solutions [20]. Numerical methods require explicit gradients and can fall into local minima, instead heuristics are better suited to large, nonlinear search spaces. Furthermore, they can also adapt dynamically to diverse situations, and address multiple objectives simultaneously. In this chapter, the following optimizing methods will be analyzed: evolutionary algorithms, particle swarm optimization and memetic optimization.

## 3.1 Evolutionary algorithms

Evolutionary algorithms (EAs) are stochastic search methods that mimic the metaphor of natural biological evolution and the social behavior of species [21]. These algorithms do not require any gradient information and typically use a set of design points called population to find the optimum result.
All the different evolutionary algorithms share the same methodology. First, the population of potential solutions is arbitrarily initialized, then it evolves toward better and better regions of the search space with the help of randomized processes of selection, mutation, and recombination. The environment delivers quality information about the search points through a fitness value, and the selection process favors those individuals of higher fitness to reproduce more often passing genetic information to the new generations than those of lower fitness. The recombination mechanism allows mixing of parental information while passing it to their descendants (offspring), and mutation introduces innovation into the population [22]. The algorithm then goes on until a termination criterion is met, such as reaching a satisfactory fitness level or a maximum number of iterations.

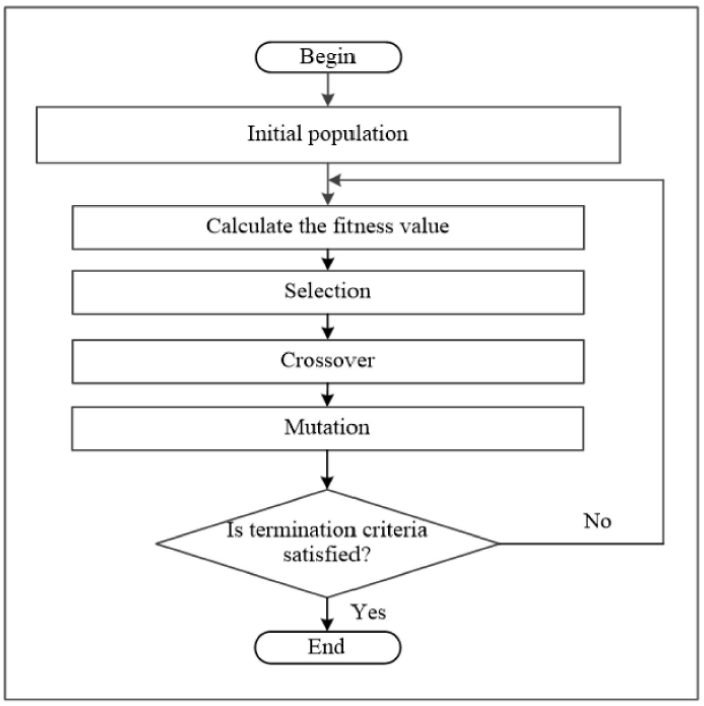


**Figure 3.1:** Flowchart of the standard genetic algorithm [6]

### 3.1.1 Initialization and encoding

This is the first step of the algorithm, where the initial population is created. Each solution is represented by a vector or a string called chromosome, which is made of several elements called genes, that contain the information. They can either be generated randomly or using some prior knowledge.
Encoding is needed to represent candidate solutions to an optimization problem in a format that can be used by the algorithm. There are different kinds of encoding:

- Real-value encoding uses continuous values
- Discrete encoding uses discrete values
- Binary encoding represents solution as bit strings
- Permutation encoding applied to sorting problems
- Tree encoding used for hierarchical structures

In this work, real-value encoding is used. An individual represents a candidate solution and it is represented by a vector of real values:

$$x = (x_1, x_2, \ldots, x_n) \qquad x_i \in \mathbb{R} \tag{3.1}$$

where $x_i$ is a gene, and $n$ is the total number of genes.

### 3.1.2 Fitness function

The fitness function evaluates the quality of a solution by computing how well it performs with respect to the optimization objective. It gives each candidate solution a score, that reflects how well the individual is performing. The evolutionary algorithm tries to minimize or maximize the function, depending on the problem request. The fitness is designed according to the specific requirements of the optimization problem and it guides the search by determining which solutions carry on into the evolution. Some examples of fitness function are:

- Distance: the objective is to bring a point $x$ close to a point target $x_T$.

$$f(x) = (x - x_T) \tag{3.2}$$

- Euclidean distance: the objective is to minimize a straight line distance between two points.

$$f(x) = \sqrt{\sum_{i=1}^{n}(x_i - x_{T,i})^2} = \|x - x_T\|_2 \tag{3.3}$$

- Minimal displacement: the objective is to minimize the movement between consecutive moves.

$$f(x) = \sum_{i=1}^{n} |x_i - x_{i-1}| \tag{3.4}$$

In a multi-objective problem, there is more than one goal to reach simultaneously. Each goal has its own score, and the final fitness function is the combination of all the fitness scores together. Some weights can be added to the single scores to prioritize certain goals. The final solution will balance the different objectives:

$$f(x) = w_1 f_1(x) + w_2 f_2(x) + \cdots + w_n f_n(x) \tag{3.5}$$

If the optimization problem has also some constraints to meet, a penalty function can be used to handle unfeasible solutions. It can be set in two ways:

- Hard constraints: if the solution does not satisfy the constraints, it is assigned a very poor fitness score, which makes it discarded from the set of possible solutions.

$$f_C(x) = \begin{cases} f(x) & \text{if } x \text{ is feasible} \\ \infty & \text{if } x \text{ is unfeasible} \end{cases} \tag{3.6}$$

  where $f_C(x)$ is the final fitness function, and $f(x)$ is the one without constraints.
  This method guarantees valid solutions, but it can slow down the optimization if many solutions are unfeasible.

- Soft constraints: when a constraint is violated, a penalty function is added to the fitness.

$$f_C(x) = f(x) + \lambda P(x) \tag{3.7}$$

  where $\lambda$ is the penalty weight which determines how strongly violations are penalized, and $P(x)$ is the penalty function.
  This method allows exploration of unfeasible solutions, but it requires careful tuning of the penalty weight $\lambda$.

### 3.1.3 Parent selection

When the evolution cycle begins, the individuals that have a higher fitness are selected to become parents. Yet, only choosing individuals with the best fitness score, completely excluding the other ones, could lead to premature convergence. Therefore, some individuals with worse fitness are also chosen to be parents maintaining diversity in the population. There are three main methods of parent selection:

- Roulette wheel selection: the individual is chosen with probability proportional to its fitness. It does not work well if one solution has a higher score compared to the

others; if all the scores are similar it will work as a random selection.

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} f(x_j)} \tag{3.8}$$

- Rank selection: it is similar to the previous one, but the population is ranked by fitness and the probability is proportional to the relative fitness rather than the absolute fitness.

$$P(x_i) = \frac{n - i + 1}{\sum_{i=1}^{n} i} \tag{3.9}$$

- Tournament selection: first a small group of individuals is randomly selected, then the best individual among them is chosen as a parent.

### 3.1.4 Recombination

Recombination, or crossover, is the procedure by which two parents are combined exchanging genetic material to create a new offspring. There are different kinds of crossovers:

- Single point: a random cross over point is selected and it is used to split the chromosome. Then, the genes that are on one of the two sides are switched.
- Double point: two random points of the chromosomes are chosen, and the genetic material is exchanged between those points.
- Uniform: each gene of the new offspring is selected randomly from one of the corresponding genes of the parents.
- Arithmetic: a linear combination function is defined to recombine the parents. It smoothly blends parent solutions instead of just switching components.

$$x_i^{offspring} = \alpha x_i^{parent1} + (1 - \alpha) x_i^{parent2} \tag{3.10}$$

Choosing $\alpha = 0.5$, the arithmetic mean of the genes' values is obtained.

### 3.1.5 Mutation

Mutation is a small, random modification in the chromosome, used to get a new solution and avoid getting stuck in local optima. It improves convergency of the solution, where recombination no longer shows advancements.
There are different types of mutation:

- Bit flip: it is used in binary encoding, it flips the bit of random genes.
- Random resetting: it is the bit flip for integer representation. A random value is chosen between the range of the possible ones, and it is assigned to a random gene.
- Swap mutation: two random genes are selected and switched.

- Scramble mutation: a subset of genes is selected and randomly shuffled.
- Inversion mutation: a subset of genes is selected and inverted.

### 3.1.6 Parallel islands

Parallel islands are used to improve diversity and increase the search convergence speed. The population is divided into several groups called islands. These islands evolve independently in parallel, following the same steps of evolution. Periodically, the islands can share genetic material through migration to exchange useful information to find the best solution. By doing so, the exploration of multiple regions of the solution space is encouraged.
When the algorithm stops, the individual with the best fitness score among the islands is selected.

### 3.1.7 Wipeout

After recombination and mutation, the offspring are created and the weakest solutions are eliminated through wipeout to make space for better results. Usually the individuals with worst cost function are the ones that are eliminated, but sometimes also some randomness is introduced.

### 3.1.8 Selection

Finally, the best individuals among the parents and the offspring are selected to proceed in the evolution. It is very important to maintain both good solutions and population diversity to guarantee a satisfactory solution.
There are different selection strategies:

- Replacement: the new created offspring replace the entire parent population.
- Elitism: a small number of best individuals from the parent generation are carried on without any gene modification. The remaining part of the population is then selected by some other criteria. This ensures that the best solutions from each generation are always carried on.
- Fitness: the fittest individuals are selected among the parents and the offspring to carry on into the next generation. This ensures that the best individuals are always maintained.

### 3.1.9 Termination

Termination is when the evolutionary algorithm stops running. There are different stopping conditions:

- Fitness score: the algorithm stops when a satisfactory solution is found, when the difference between the individual fitness score and the target fitness score is below a predefined threshold:
$$|f(x_{best}) - f_{target}| < \epsilon \tag{3.11}$$
where $f(x_{best})$ is the fitness score of the best individual, $f_{target}$ is a predefined target score, and $\epsilon$ is a small threshold value.

- Convergence: the algorithm stops when there is a little or no change in the fitness score of the population over the generation:
$$|f(x^g_{best}) - f(x^{g-1}_{best})| < \delta \tag{3.12}$$
where $f(x^g_{best})$ is the best fitness score at the current generation, $f(x^{g-1}_{best})$ is the best fitness score at the previous generation, and $\delta$ is a small fitness threshold value.

- Maximum number of generations: the algorithm stops after a predefined number of generation is reached:
$$g > g_{max} \tag{3.13}$$
where $g$ is the current generation number, and $g_{max}$ is the selected maximum number of generations.

- Timeout: the algorithm stops after a predefined time limit is reached:
$$t > t_{limit} \tag{3.14}$$
where $t$ is the current time, and $t_{limit}$ is the time limit.

## 3.2 Particle swarm optimization

The particle swarm algorithm is a stochastic optimization technique inspired by the social behavior of a flock of birds or a school of fish migrating and trying to reach an unknown destination. The algorithm uses shared information among particles to drive the search to an optimal solution [23]. Each candidate solution is represented by a particle that moves through the search space by modifying its velocity and position based on its own experience and the knowledge of the swarm. In this way, the particles are able to dynamically balance exploration and exploitation, and converge towards optimal solutions more efficiently.

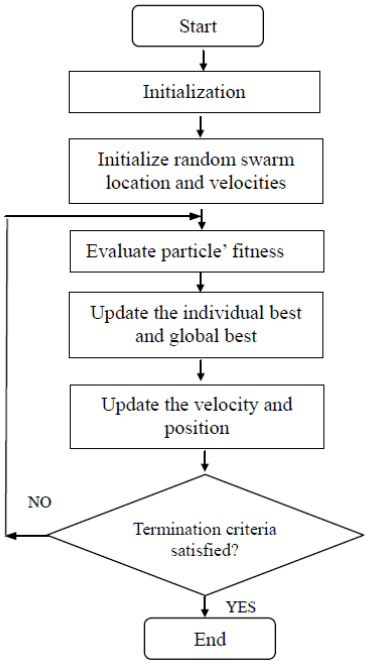


**Figure 3.2:** Flowchart of the standard particle swarm optimization algorithm [7]

### 3.2.1 Initialization

The population of particles is initialized, each one representing a candidate solution. In the search space, each particle has its own position and velocity. In a n-dimensional search space, position and velocity of particle $j$ are respectively $x_j = (x_{j1}, x_{j2}, \ldots, x_{jn})$ and $v_j = (v_{j1}, v_{j2}, \ldots, v_{jn})$. For each particle, the personal best $x_{pers,best}$ is evaluated, and then the global best $x_{glob,best}$ is evaluated among all the particles.

### 3.2.2 Fitness function

The fitness function of a particle measures the quality of the solution by evaluating how well the particle's position performs relative to the optimization objective. The goal of PSO is to find the position that minimizes (or maximizes) the fitness function.

### 3.2.3 Update

At each iteration, using the current velocity and the distance from the personal and global best, position and velocity of the particles are updated [24].
The updated velocity of each particle is calculated as:

$$v_{ij}^{k+1} = w \cdot v_{ij}^{k} + C_p r_p (x_{pers,best,ij}^{k} - x_{ij}^{k}) + C_g r_g (x_{glob,best,ij}^{k} - x_{ij}^{k}) \qquad i = 1, 2, \ldots, N \quad j = 1, 2, \ldots, p \tag{3.15}$$

where $k$ is the iteration count, $v_{ij}^{k+1}$ is the velocity of the $j^{th}$ particle of the $i^{th}$ variable at the $(k+1)^{th}$ iteration, $w$ is the inertia weight, $x_{ij}^{k}$ is the position of the $j^{th}$ particle of

the $i^{th}$ variable at the $k^{th}$ iteration, $C_p$ and $C_g$ are the cognitive and social acceleration coefficients, $N$ is the total number of variables, $x^k_{pers,best,ij}$ is the personal best position of the $j^{th}$ particle of the $i^{th}$ variable at $k^{th}$ iteration, $x^k_{glob,best,ij}$ is the global best value of $i^{th}$ variable until $kth$ iteration, and $r_p$ and $r_g$ are uniformly distributed random numbers.
The updated position of each particle is calculated as:

$$x^{k+1}_{ik} = x^k_{ik} + v^{k+1}_{ik} \qquad i = 1, 2, \ldots, N \qquad j = 1, 2, \ldots, p \tag{3.16}$$

where $x^k_{ik}$ is the position of the $j^{th}$ particle of the $i^{th}$ variable at the $k^{th}$ iteration, and $v^{k+1}_{ik}$ is the updated velocity.

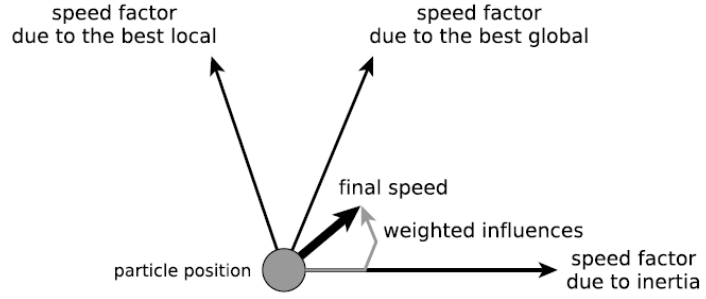


**Figure 3.3:** Influence of factors on the particle speed in PSO [8]

### 3.2.4 Termination

When the algorithm stops, the best known solution is kept.
There are different termination criteria:

- Fitness score: the algorithm stops when a satisfactory solution is found, when the difference between the individual fitness score and the target fitness score is lower than a predefined threshold.

  $$|f(x^k_{glob,best}) - f_{target}| < \epsilon \tag{3.17}$$

  where $f(x^k_{glob,best})$ is the fitness score of the best position at iteration $k$, $f_{target}$ is a predefined target score, and $\epsilon$ is a small threshold value.

- Fitness convergence: the algorithm stops when the fitness value does not improve after a certain number of iterations:

  $$|f(x^k_{glob,best}) - f(x^{k-m}_{glob,best})| < \delta \tag{3.18}$$

  where $f(x^k_{glob,best})$ is the fitness score of the best solution at iteration $k$, $f(x^{k-m}_{glob,best})$ is the fitness score of the best solution found $m$ iterations earlier, and $\delta$ is a small threshold that determines if the improvement is too small.

- Velocity convergence: the algorithm stops if the velocity change between iterations is very small, i.e. if the average norm of the velocities is lower than a threshold:

$$\frac{1}{p}\sum_{j=1}^{p}\|v_j^k\| < \delta \tag{3.19}$$

where $\|v_j^k\|$ is the velocity vector norm, $p$ is the number of particles and $\delta$ is a velocity threshold.

- Maximum number of iterations: the algorithm stops when a predefined number of iterations is reached:

$$k > k_{max} \tag{3.20}$$

where $k$ is the current iteration and $k_{max}$ is the maximum number of iterations.

- Timeout: the algorithm stops when a predefined time limit is reached:

$$t > t_{limit} \tag{3.21}$$

where $t$ is the current time and $t_{limit}$ is the time limit.

## 3.3 Memetic algorithms

Memetic optimization methods are advanced evolutionary algorithms, in which all chromosomes and offspring are allowed to gain some experience, through a local search, before being involved in the next evolutionary process [25].
During evolution, after new candidate solutions are generated, they are refined using heuristic or gradient based methods. In this way, new promising solutions are discovered and also improved, leading to even greater precision.
Thanks to this hybrid search that balances global exploration with local exploitation, efficiency is improved and faster convergence and higher quality solutions are achieved.
Memetic algorithms follow the same phases as evolutionary algorithms, with the addition of the local search [26]. First the population is initialized, then a fitness function is calculated for each individual. Subsequently, the best candidates are selected and they mate to create offspring through the crossover and mutation phases. Now, the local search is introduced, the most promising solutions undergo refinement to enhance the solution quality. The population is then updated, and the cycle is repeated until a termination criterion is found.
Several types of local search techniques exist, such as:

- Hill climb: it is an iterative algorithm that starts with an arbitrary solution, then it moves incrementally to the best neighboring state trying to find a better solution. If one is found, the algorithm goes on making another incremental change, until no further improvements can be found [27].

- Simulated annealing: it is a stochastic local search, used especially when many local minima are present. It works in a similar way as hill climbing, but sometimes with a probability that decreases over time, it also accepts worse solutions [28].

- Gradient based: it is commonly used when the fitness function is differentiable, for continuous optimization problems. This method uses derivatives (gradients) to guide the search towards an optimal solution [29]. At each iteration the gradient of the cost function $\nabla f(x)$ is calculated, and the current position is updated moving in the opposite direction with a step size determined by the learning rate $\alpha$.

$$x_{t+1} = x_t - \alpha \nabla f(x_t) \tag{3.22}$$

- Greedy search: it determines the locally optimal choice at each step, without backtracking or accepting worse solutions. It is a fast search, but it does not always guarantee an optimal solution [30].

# Chapter 4

# Simulation framework

## 4.1 ROS

ROS (Robot Operating System) [31] is an open-source framework designed to facilitate the development of robotic applications. Even if the acronym might be misleading, it is not an operating system, but rather a middle-ware that functions as an intermediate layer between the operating system and robotic applications.

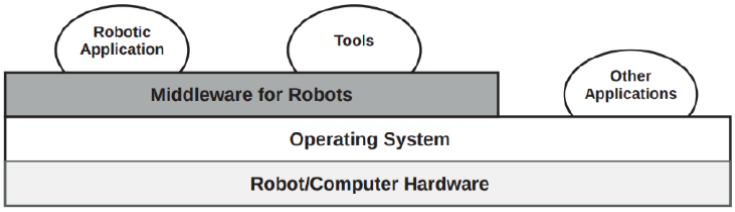


**Figure 4.1:** Software layers in a robot [9]

Initially developed by Willow Garage in 2007, ROS has become one of the most widely used platforms for robotics research and development. It provides a flexible and modular architecture that simplifies the integration of complex robotic systems by offering standardized communication protocols, tools, and libraries. The first version, ROS 1, was released in 2010 with modular design and ease of use. Then, in 2017, a second version was released, ROS 2, which aimed at improving some previous limitations, like real time handling.
One of the key elements of ROS is its distributed computing architecture. It utilizes a graph-like structure where nodes, which are individual software processes, communicate with each other through topics, services, and actions. This communication is facilitated by a publish-subscribe messaging (pub-sub) system, allowing nodes to send and receive data efficiently [32].
This structure gives ROS several advantages [33]:

- Modularity and reusability: difficult robotics system can be divided into independent nodes. ROS packages can be integrated into new projects, and components can be easily tested and reused.

- Peer-to-peer: after a discovery process, a central server is not needed, because ROS nodes communicate with each other directly in a decentralized way.
- Portability: ROS can run on several operating systems, including Linux and Windows.
- Standardize communication: ROS provides an organized communication structure, made of messages, topics, services and actions.
- Flexibility: ROS can be used and adapted to a wide range of robotics applications, from industrial automation, to space and underwater.
- Free and open source: ROS platform is free and open source, and it is used by a large number of developers. A lot of already built packages, documentation and tutorials are available.
- Multilingual: ROS is designed to work with several programming languages: C++, Python, and Lisp.
- Graphics: ROS includes powerful visualization tools, such as RViz and Gazebo, to simulate robot performance.

### 4.1.1 Architecture

The ROS (Robot Operating System) architecture design provides modular and flexible development of robotic systems. The communication model supported by ROS is distributed and decentralized, facilitating the construction of complex robotic systems. The ROS architecture is composed by nodes, that communicate with each other to perform a specific task, and are managed by a ROS master. They can send messages to a specific topic through a publisher and process incoming messages through a subscriber.
The fundamental concepts of the ROS implementation are nodes, messages, topics, services and actions.

#### Nodes

A node is a fundamental building block of ROS programs, uniquely defined by its name. It is one computational unit, which represents a process that performs a specific task. Nodes are independent processes that can be distributed across multiple machines; the possibility of testing them individually makes the development and debugging more efficient.
They can communicate with other nodes to exchange information and coordinate tasks. Once nodes are discovered, they can directly exchange data with each other without the intervention of the ROS Master. This peer-to-peer communication helps to minimize latency and improves performance in distributed systems.
Nodes that publish messages to a topic do not need to know the subscribers, and subscribers do not need to know the publisher. This makes the communication system highly decentralized and scalable.

### Master

The ROS Master is a central part of the architecture because it facilitates communication between nodes. It manages all the ROS nodes that are connected to the network, keeping track of them and providing a name service to discover topics, services and parameters. The ROS Master does not handle the actual data transfer but helps with the discovery of nodes and provides the establishment of initial communication between them. When a node wants to communicate, it requests the Master to acquire information about topics or services that are available. Once nodes have discovered one another, they establish direct links and share data peer-to-peer.

### Messages

Massages are used by the nodes to communicate with each other and contain the actual content that is exchanged. A node publishes a message to a given topic, and all the nodes that are interested in that area will subscribe to the appropriate topic. A message contains the data type transmitted over a topic; they can be integers, floats, strings or others.
A message file is a simple text file that contains the data structure of the message [10]. For example, a point in the space is described by the message in Figure 4.4.

```
# This contains the position of a point in free space
float64 x
float64 y
float64 z
```

**Figure 4.2:** Massage of a point in the space [10]

### Topics

ROS topics use the asynchronous publish-subscribe model, where publishers broadcast messages to a topic, and subscribers receive messages from the topic without knowing the publisher. Topics are the channels that are used for this communication and that identify the particular data exchange. This makes the system more modular and adaptable, since the direct connection between nodes is not required to be maintained. Multiple nodes can subscribe to a topic and receive data at the same time, and a single node can publish and subscribe to multiple topics.

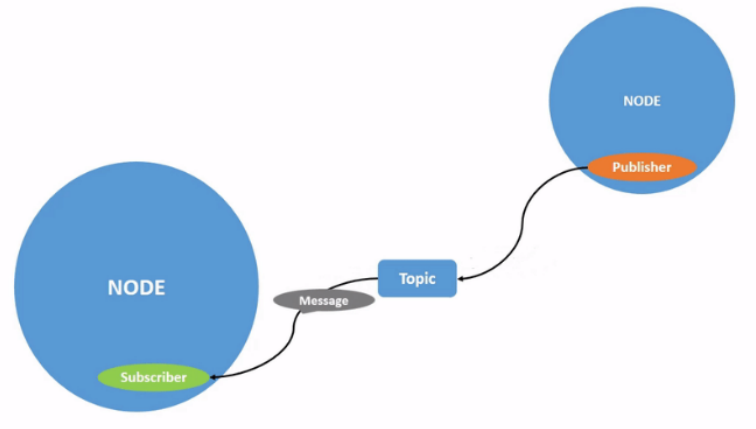


**Figure 4.3:** ROS topic [11]

### Services

While topics work for unidirectional communication, services adopt a synchronous request-response communication model. One node sends a request and another node responds, therefore a pair of messages is used, one for the request and one for the response.

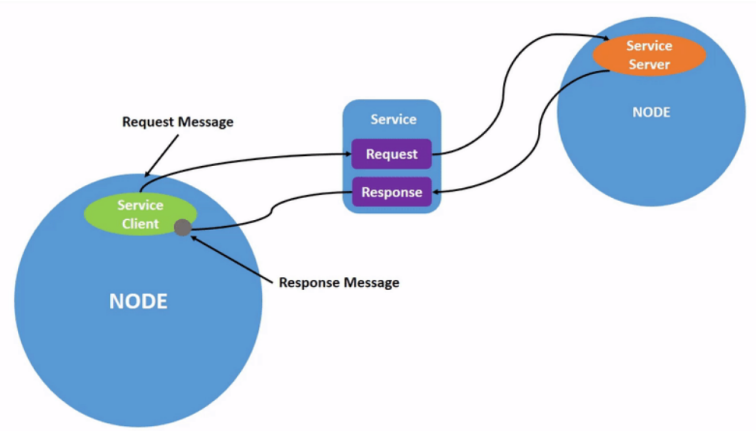


**Figure 4.4:** ROS service [12]

### Actions

Actions are similar to services but they work for tasks that can take a long time to run. The communication is asynchronous, which means that a node can be sending a message and meanwhile receives responses.

### Publisher and subscriber

Publishers and subscribers are very important mechanisms for the nodes to communicate using topics. With a publisher, a node can send messages to a specific topic, while with a subscriber a node can listen to the topic and process incoming messages. Thanks to this method, different parts of the robotic system can communicate asynchronously.

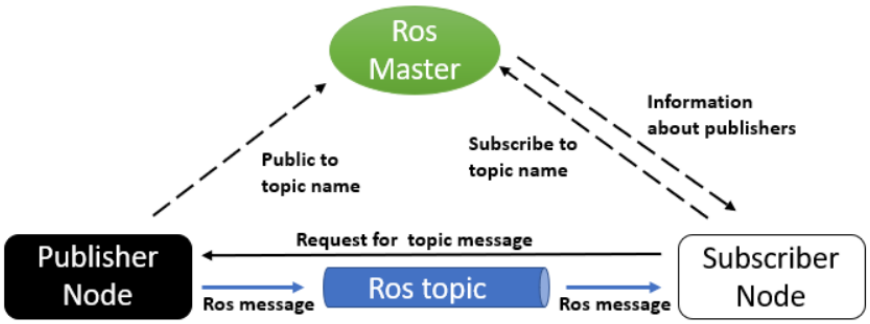


**Figure 4.5:** ROS publisher and subscriber [13]

### Packages

A ROS package [34] is the basic unit of organization in ROS. Each package contains nodes, configuration files, libraries, scripts, and message definitions. They can be built independently and used in the ROS environment. Inside a ROS package it is possible to find:

- Launch folder: it contains the launch files (.launch) that make possible to launch multiple ROS nodes at one and establish connection between nodes.
- Src folder: it contains the source code of the package (typically C++ or Python), where ROS nodes and other scripts are implemented.
- package.xml: it contains meta information about the package, including its name, version, description, maintainer, license, dependencies, and more.
- CMakeLists.txt: it contains the building instructions for the package.

### Workspace

The ROS workspace contains all the ROS packages. Workspaces allow for the organization of multiple packages and nodes.

## 4.1.2 ROS commands

ROS commands are a set of tools that developers use to interact with the ROS environment. There are several commands, and each of them has a specific purpose.
Some of the most used are:

- `roscore`: it initializes the ROS system core components, such as the ROS master, the parameter server and the logging system.
- `rosrun`: it executes a selected node from a given package.
- `roslaunch`: it launches multiple nodes and configures them using a single launch file.

- `rostopic`: it is used to get information about a given topic.
- `rosservice`: it interacts with services in the system, by calling, listing and getting information about them.
- `rosnode`: it shows the current node information.

### 4.1.3 URDF

URDF (Unified Robot Description Format) is used to describe robot models in ROS. It describes the robot physical components, such as joints, links, and sensors in a unified way. The file has xml format and it allows the robot to interact with visualization tools and simulation environments such as RViz, MoveIt and Gazebo.
The key elements of the file are:

- `<robot>`: it includes the whole robot description.
- `<link>`: it defines the robot's components.
- `<joint>`: it defines how two links are connected and how they move in relation to each other.
- `<visual>`: it describes the shape of the link.
- `<collision>`: it defines the collision geometry of the link.

## 4.2 RViz

RViz (Robot Visualization) is a 3D visualization tool in ROS that allows users to display sensor data, such as LiDAR, camera feeds, and point clouds, robot models (URDF) with joints and links, and planning results in an interactive environment. It helps to understand robot kinematics showing real-time transformations (TF frames), and it has an interactive interface for setting target poses and verify motion planning results. It also supports marker visualization, which is used for displaying objects and paths. It is widely used for debugging, simulation, and monitoring real-time robotic operations.

## 4.3 MoveIt

MoveIt [35] is a motion planning framework in ROS designed for robotic manipulation and kinematics. It provides tools for inverse kinematics, collision avoidance, and path planning, making it essential for robotic arms and mobile manipulators. It computes the forward and inverse kinematics, it generates collision-free trajectories, avoiding obstacles using OctoMap and planning scene monitoring. It also supports grasp planning and execution for picking and placing objects.
Usually MoveIt is used with RViz to visualize planned paths before the execution.

# Chapter 5

# Racer-5 COBOT

## 5.1 Robotic manipulators

An industrial robot is a self-operating machine that was geared for use in industries where assembly, welding, painting, and material handling were needed. Its purpose is to maximize productivity efficiency, accuracy, and safety within repetitive or hazardous tasks which could be performed non-stop without tiring.
Standard industrial robots are made of articulated arms, Cartesian Robots, anthropomorphic robots, and SCARA Robots, all of which have advanced control and sensor systems for precise movement. In recent years, the rise of collaborative robots (cobots) has made it possible for humans and robots to work alongside each other safely in the same environment, boosting adaptability and enhancing productivity for workers.
In this chapter, the features and performance of the Racer-5 COBOT are examined.

## 5.2 Robot overview

The Racer-5 COBOT is a collaborative robot developed by Comau, designed to merge the speed and precision of industrial robots with the safety features demanded for the interaction between humans and robots. It is capable of operating in both industrial and collaborative modes, thanks to its ability to balance safety and productivity.

## 5.3 Technical specifications

The Racer-5 COBOT has six degrees of freedom. Its key specifications involve:

- Payload Capacity: 5 kg
- Maximum Reach: 809 mm
- Repeatability: $\pm 0.03$ mm
- Collaborative/non-collaborative speed switch


**Figure 5.1:** Racer-5 COBOT by Comau [14]

- Speed: Up to 500 mm/s cartesian speed in collaborative mode
- Speed: Up to 6 m/s cartesian speed in non-collaborative mode
- Weight: Approximately 34 kg
- Safe Collision Detection function - PL d CAT. 3 certified by TÜV Süd

## 5.4 Robot kinematics

The Racer5-COBOT is a 6 DOFs manipulator, composed of six revolute joints that allow rotational movement. The kinematic chain is illustrated in Figure 5.2. The base is fixed and it is represented by a white rectangle, while the joints are represented by grey cylinders. Each cylinder is oriented to match the orientation of its corresponding joint: for vertical joints, the cylinder is shown as a rectangle, and for horizontal joints, it is represented as a circle. A reference frame that follows the Denavit-Hartenberg (DH) convention is assigned to each joint.

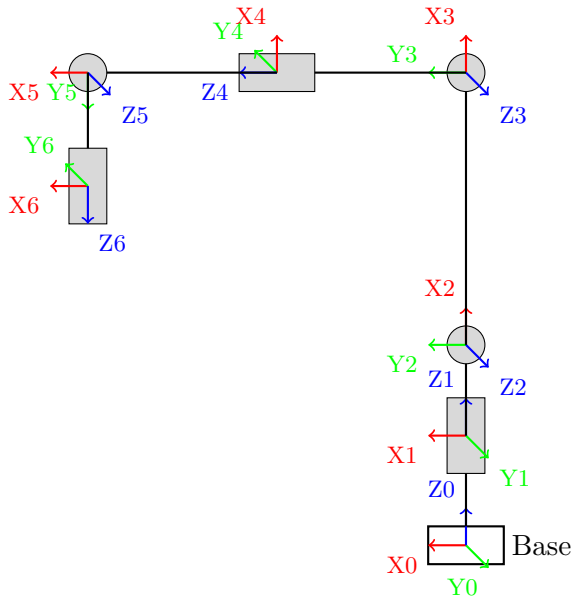


**Figure 5.2:** Kinematic chain of Racer5-COBOT by Comau

The Denavit-Hartenberg (DH) parameters are used to model the kinematics of the robot.

| **Link** | **a (m)** | $\alpha$ **(rad)** | **d (m)** | $\theta$ **(rad)** |
|---|---|---|---|---|
| 1 | 0 | 0 | 0.1895 | $\theta_1$ |
| 2 | 0.1755 | $-\frac{\pi}{2}$ | 0 | $\theta_2$ |
| 3 | 0.37 | 0 | 0 | $\theta_3$ |
| 4 | 0 | $-\frac{\pi}{2}$ | 0.121 | $\theta_4$ |
| 5 | 0 | $\frac{\pi}{2}$ | 0 | $\theta_5$ |
| 6 | 0 | $-\frac{\pi}{2}$ | 0.0665 | $\theta_6$ |

**Table 5.1:** Denavit-Hartenberg Parameters

To represent the relative motion between each link, the transformation matrices are derived from the DH parameters table. The overall pose of the end effector with respect to the base frame is then given by the total transformation matrix, found by multiplying those individual matrices sequentially.
The workspace of the Racer5-COBOT is defined by a semi-spherical volume determined by its joint limits and link lengths, within which the end effector can reach different positions and orientations. It is illustrated in Figure 5.3.

## 5.5 End-effector

The end effector is the final component of the kinematic chain and directly interacts with the environment to perform the required tasks. It has an asymmetric structure and it is composed by two distinct geometric shapes: a cubic box and a rectangular box. The cubic

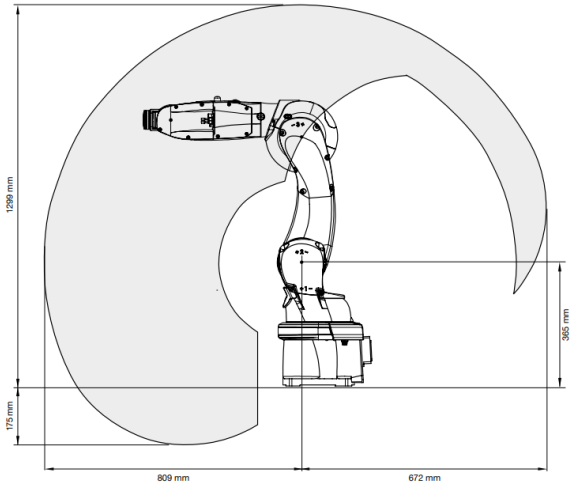


**Figure 5.3:** Racer5-COBOT workspace [14]

one is attached to the last link and it has dimensions $x = 0.05, y = 0.05, z = 0.05$, while the rectangular one is attached to the cube and has dimensions $x = 0.35, y = 0.1, z = 0.04$.

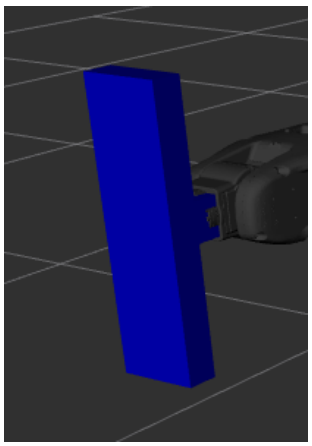

**Figure 5.4:** End effector

## 5.6 Applications

Some of the common use cases for the Racer-5 COBOT are:

- Assembly: precise handling of small components in electronics and automotive industries
- Material handling: moving, storing, and managing materials or products within a facility or supply chain efficiently
- Machine Tending: loading and unloading materials or parts into machines for automated manufacturing

- Pick and Place: automation process of picking up an object from one location and placing it in another location

Its ability to work safely next to humans makes it ideal for environments where automation needs to be flexible, responsive and safer.

# Chapter 6

# BioIK

BioIK is a memetic inverse kinematics solver developed for the motion planning framework MoveIt and the robot operating system ROS. The memetic algorithm uses a combination of evolutionary optimization, particle swarm optimization, and gradient based methods. Differently from other kinematics solvers, it supports kinematic trees with multiple end effectors.
At first, the TAMS research group developed in C# for the Unity3D game engine a multi-objective inverse kinematics solver based on evolutionary algorithms [36]. Then, the algorithm was ported to C++ (with few minor algorithmic changes that did not modify its general structure) and integrated into MoveIt as an inverse kinematics plugin [37]. Finally, it was adapted specifically to the requirements of robotic systems.

## 6.1 Algorithm overview

The algorithm is based on genetic optimization and combines evolutionary optimization methods, particle swarm optimization, and memetic algorithms.

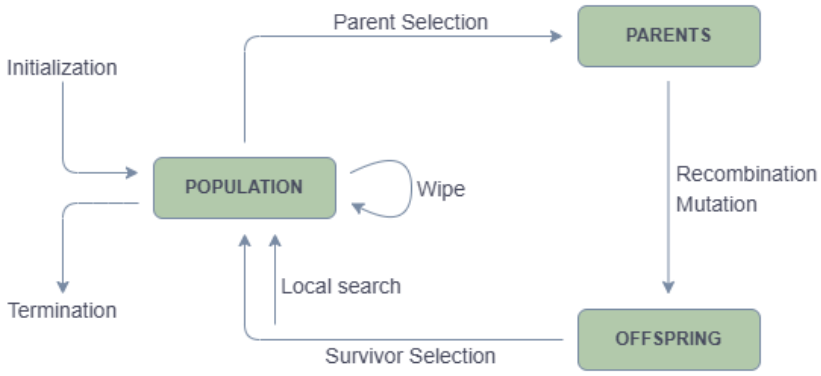


**Figure 6.1:** BioIK algorithm flowchart

### 6.1.1 Encoding

Each individual is represented by a genotype $x$, which is composed of $n$ genes. Each gene is denoted with $x_i$ and represents the i-th joint variable of a kinematic chain with $n$ degrees of freedom.

$$x = (x_1|x_2|x_3|\dots|x_{n-1}|x_n) \tag{6.1}$$

Each genome then encodes a particular joint configuration $\Theta$ of a joint-space robot pose. In the first version of the algorithm, 32 bit single precision numbers were used, while in the second one they were replaced with 64 bit double precision.
Since robots have joints limits that can not be exceeded, each joint has a minimum and a maximum value, respectively $x_i^{\min}$ and $x_i^{\max}$. A gene is clipped if it exceeds these values. Clipping ensures feasibility, reduces search space complexity and keeps calculations efficient. Finally, each gene also has a gene span $s_i$, which scales the magnitude of mutations by representing the range over which the gene can change.

### 6.1.2 Fitness function

The fitness is a function that evaluates how good an individual performs in solving the given problem. It assigns a numerical score to each candidate solution, based on its ability to satisfy the objectives of the problem (the lowest the score, the better the solution).
At each generation the fitness is evaluated and the individuals are ranked based on it, and selected according to the best scores.
An example of a fitness function is how close the end-effector gets to the target pose, calculated as the squared distance between the actual pose and the wanted pose. Another fitness function could be the distance of the joints from the joint limits, penalizing configurations that are close to the constraints and favoring others.
In multi-objective optimization, as is the case in this study, for each individual multiple goals are present. They can be weighted based on relevance, and the final fitness is the sum of these weighted goals.

### 6.1.3 Parent selection

The parent selection strategy determines how individuals from the current population are chosen to contribute genetic material to the next generation. In this case it is based on rank selection. Given that the population is ordered according to their fitness value, the probability of choosing a solution as a parent is:

$$p(i) = \frac{N - i + 1}{\sum_{i=1}^{N} i} \tag{6.2}$$

where $N$ is the total number of individuals in the population and $i$ is the rank of an individual. Each individual is assigned a probability according to the formula above, then a cumulative distribution is created by summing up the probabilities up to and including that individual. An uniformly distributed random value $r$ is chosen between 0.0 and 1.0,

and the first individual whose cumulative probability exceeds it is chosen as a parent from the mating pool $\Gamma$ (which is initially the whole population). The probabilities sum up to 1 and are normalized within the range $[0.0, 1.0]$, the parent selection operator $S_P$ is

$$S_P : P_{\{1,2\}} \leftarrow p(\Gamma_i) = \frac{N - i + 1}{\sum_{i=1}^{N} i} \tag{6.3}$$

The selected individual $\Gamma_i$ is chosen to be a parent, and it will be involved in the recombination phase of the genetic algorithm, where it will potentially combine its genetic material with another individual to produce offspring. With this approach, the likelihood distribution continuously decreases with the quality of an individual, it is efficient because it adapts effectively to any population size and does not depend on the specific distribution of fitness values. Individuals with higher fitness have a greater chance of being selected to reproduce, but there is also randomness to ensure diversity in the population.

### 6.1.4 Reproduction

Reproduction is the mechanism by which new individuals are generated. It combines the genetic information of parent individuals to explore new possible solutions while maintaining diversity in the population. It incorporates two mechanisms: recombination (crossover), which mixes genes from two parents to produce a new individual, and mutation, which introduces random changes to stimulate exploration and prevent premature convergence. The combined mechanisms ensure that the algorithm gets an equilibrium between exploitation (refining good solutions) and exploration (searching in new areas of the solution space).

#### Recombination

The recombination operator combines information from two parents to create new offspring solutions.
The gradient information from each parent is blended together, guiding the evolution towards more promising directions. The new gradient is

$$g_{new} = (1 - \alpha) \cdot g_{parent1} + \alpha \cdot g_{parent2} \tag{6.4}$$

where $\alpha$ is a mixing factor.
The child genomes are mainly inherit from one parent, but a small fraction is received from the other parent.

#### Mutation

The process of mutation introduces slight, random variations to the genotype of individuals. This random procedure is needed to maintain genetic diversity, such that the algorithm can explore new solutions of the search space and prevent premature convergence to local optima. Mutation rate, which determines how frequently these changes take place, is normally kept low to avoid excessive disruption to the evolving solutions.

Each one of the genes $x_i$ mutates individually in the mutation process, which is made of two parts: first a stochastic disturbance proportional to the gene's allowed range, then a refinement based on gradient information.
The dynamic mutation rate $m$ is defined as:

$$m = 2^{k-23} \tag{6.5}$$

where $k$ is a random integer in the range $[0; 15]$. The mutation rate is typically very small. The range of each gene is defined as $\Delta x = x_{max} - x_{min}$. Multiplying the mutation factor by $\Delta x$, the mutation is appropriately scaled and normalized to the gene's range. Then, to determine the magnitude and direction of the mutation, a random Gaussian variable $r$ is used. The perturbation is then defined as:

$$p = r \cdot m \cdot \Delta x \tag{6.6}$$

The perturbation factor is then added to the original gene:

$$x' = x + p = x + r \cdot m \cdot \Delta x \tag{6.7}$$

Then, the algorithm follows a mechanism similar to particle swarm optimization, where information about previous good solutions is used to accelerate convergence. To favor the mutation towards promising directions, the gradient information from the two parents is added to the algorithm. $g_1$ is the gradient value for the gene from parent number 1, while $g_2$ is from parent number 2. To combine the two gradients, a linear interpolation is performed:

$$\gamma = (1 - f) \cdot g_1 + f \cdot g_2 \tag{6.8}$$

where $f$ can be equal to 0.0 or 0.2 depending on the index of the offspring.
Finally the combined gradient is multiplied by a momentum inspired factor, similar to velocity updates in Particle Swarm Optimization, PSO, $\lambda \in \{0, 1, 2\}$:

$$c\_grad = \lambda \cdot \gamma \tag{6.9}$$

To prevent losing direction, the value of $\lambda$ is chosen only once per individual. This is inspired by PSO's velocity update mechanism, where direction and magnitude are maintained rather than being randomly reassigned at each step.
Hence, the final gene is defined as:

$$x' = x + p + c\_grad = x + r \cdot m \cdot \Delta x + \lambda \cdot \gamma \tag{6.10}$$

The gene needs to satisfy the gene's limits, so the final result is clamped:

$$x' : \begin{cases} x_{\min} & x + r \cdot m \cdot \Delta x + \lambda \cdot \gamma < x_{\min} \\ x + r \cdot m \cdot \Delta x + \lambda \cdot \gamma & x_{\min} \leq x + r \cdot m \cdot \Delta x + \lambda \cdot \gamma \leq x_{\max} \\ x_{\max} & x + r \cdot m \cdot \Delta x + \lambda \cdot \gamma > x_{\max} \end{cases} \tag{6.11}$$

The algorithm also updates an associated gradient for the offspring using a weighted mix, thus the mutation contributes to the next generation's direction:

$$\lambda_{child} = (1 - \beta) \cdot \gamma + \beta \cdot (x' - x) \qquad \beta = 0.3 \tag{6.12}$$

This formulation captures the double nature of the mutation: a stochastic component that introduces variability, scaled by the gene's range, and an additional gradient term that directs the search towards regions of higher fitness. Together, these operations allow the algorithm to balance exploration with exploitation, while ensuring that all gene values remain within their valid domains.

### 6.1.5 Survivor selection

This part of the algorithm is used to determine, based on the fitness score, which are the best individuals, from both the parents and the children, that will survive for the next generation.
The algorithm can handle both primary and secondary goals separately. When the secondary goals are present, the individuals are first filtered based on these criteria. Then, the remaining ones are selected according to the primary goals. However, primary and secondary objectives could conflict and cancel each other out, if too many or too less individuals survive the pre-selection. To resolve this issue, a random number of survivors during pre-selection is chosen by the algorithm, guaranteeing a more balanced optimization process.

### 6.1.6 Initialization

Each individual is initialized by assigning to its genes random values within the range of the domain boundaries of the search space dimensions. However, a totally random initialization, would mean to find completely different solutions each time. To avoid this problem, the algorithm keeps exactly one solution based on the current joint configuration of the kinematic model ( $x^1 = (\theta_1|\theta_2|\dots|\theta_N)$ ), while the others are chosen randomly ( $x^{2,\dots,n} = (random_1|random_2|\dots|random_N)$ ). A totally random restart happens when solutions are only needed occasionally and when the algorithm gets stuck in a local minima. The algorithm ensures that values are assigned only to active joint variables, storing inactive joint positions in a temporary buffer that preserves their original values. Thus, the focus is on optimizing the degrees of freedom relevant to the IK problem. Moreover, space is created for offspring that will be generated and mutated in the following steps. The

gene values need to stay in the valid joint ranges, so the algorithm initializes minimum, maximum, and span values for each active joint.

### 6.1.7 Termination

The algorithm terminates when a satisfactory solution is identified, or when a timeout is reached. A solution is considered acceptable if it is below an established threshold.
For pose goal, position goal and orientation goal, the termination condition is defined by the Cartesian accuracy of the end effector final pose. Let the current position and the orientation errors be denoted respectively by $E_P$ and $E_O$, and the maximum allowed errors be denoted by $E_P^{max}$ and $E_O^{max}$. The search terminates when $E_P \leq E_P^{max}$ for position, $E_O \leq E_O^{max}$ for orientation, and $E_P \leq E_P^{max} \wedge E_O \leq E_O^{max}$ for pose.
For the other goal types, the fitness score is compared with a maximum allowed error and the solution that respects the constraints terminates the algorithm.

### 6.1.8 Islands

The population is distributed across four parallel islands, and on each one of them evolution is run independently. The process is stopped on all the islands when a solution is found, and the best result is picked. This helps to avoid premature convergence, improves diversity and enhances parallelism.

### 6.1.9 Species and wipeouts

Within each island, individuals are grouped into two species based on a similarity measure. Two individuals x and y belong to the same species if: $\delta(x, y) \leq \delta_0$, where $\delta$ is a distance metric and $\delta_0$ is a predefined threshold. Each species evolves following the evolutionary process, exploring different parts of the solution space. If for some generations a specie does not improve its fitness or if it fails a probabilistic test, it is wiped out and the individuals' genes and gradients are reinitialized with random variables. Only the species that are less fit can be subject to wipeout. The best solution found along all species is constantly updated with the fitness of the best individual of each specie as the search goes on.
Wipeout is useful to avoid local minima and help convergence to a globally optimal solution.

### 6.1.10 Memetic optimization

The genetic algorithm is then combined with a local search method to enhance and refine the solution obtained by genetic evolution.
For fast local search, gradient based optimization is run on the best individual of each species, after running evolution for a number of generations. Gradients are computed by numerically differentiating the fitness function with respect to the joint variables. The used method is the custom quadratic gradient descent method, with quadratic step size approximation.
The algorithm uses a differentiation step size *dp*, which is very small and with random sign,

to ensure that the gradient represents a small change in the fitness. Then, it is normalized to ensure that it does not become too large or too small. The new position is evaluated for fitness, and if it provides better fitness, the individual's genes are updated.
The local search stops when it does not improve the solution anymore, or if a maximum number of iteration is reached. Afterwards, evolution proceeds.

## 6.2 Goal classes

The goal types are C++ classes and they share a base class called *Goal*. The information is exchanged with the solver with a *GoalContext* object. Two methods are shared by all of the classes: *evaluate()* and *describe()*. The first one calculates how well the current configuration meets the goal and it is called after each change of joint positions. The second one indicates which joints and links are affected by the goal.
It is also possible to add new goal types, not included in the BioIK package.
In the developed algorithm, the following types of goal were used.

### 6.2.1 Position Goal

The position goal aims to align the end effector position with the target position, using a cost function given by the squared distance between the two positions.

$$cost_p = \|P_E - P_T\|^2 \tag{6.13}$$

where $P_E$ is the current end effector position and $P_T$ is the target position.

### 6.2.2 Orientation Goal

The orientation goal aims to align the end effector orientation with the target orientation, using a cost function that is the minimum squared distance between two rotation quaternions.

$$cost_o = min(\|Q_T - Q_E\|^2, \|Q_T + Q_E\|^2) \tag{6.14}$$

where $Q_E$ is the current end effector rotation quaternion and $Q_T$ is the target rotation quaternion.

### 6.2.3 Custom Partial Orientation Goal

This goal class was externally added and implemented. It aims to align the end effector orientations on the $x$ and $y$ axes (leaving the one on the $z$ axis unconstrained). The cost function is the sum of the squared distance between the angles on the $x$ axis (roll) and the squared distance between the angles on the $y$ axis (pitch).

$$cost_{po} = \|Roll_E - Roll_T\|^2 + \|Pitch_E - Pitch_T\|^2$$

where $Roll_E$ is the end effector orientation about the $x$ axis, $Roll_T$ is the target orientation about the $x$ axis, $Pitch_E$ is the end effector orientation about the $y$ axis and $Pitch_T$ is the target orientation about the $y$ axis.

### 6.2.4 Avoid Joint Limits Goal

The avoid joint limits goal aims to keep the joint variables within the center half of the joint limits. The cost function, penalizing the positions close to the joint limits, is given by:

$$cost_l = \sum_{i=1}^{N}((|j_i - \frac{h_i + l_i}{2}| \cdot 2 - \frac{h_i - l_i}{2})^2) \tag{6.15}$$

where $j_i$ is the current joint position, $N$ is the number of active joint variables, $h_i$ is the upper joint limit and $l$ is the lower joint limit.

### 6.2.5 Minimal Displacement Goal

The minimal displacement goal aims to keep the joint variables as close as possible to the last robot pose, using a cost function that is the squared distance between the current and previous joint position.

$$cost_d = \|j - k\|^2 \tag{6.16}$$

where $j$ is the current joint position and $k$ is the previous joint position.

# Chapter 7

# Algorithm and method

## 7.1 Task

The main objective of the thesis is to let the end effector of the Racer-5 COBOT reach a specified full target position $(x, y, z)$, and a partial target orientation. The orientation constraints are the following:

- the orientation angles about the $x$ and $y$ axes are fixed to target values,
- the orientation angle about the $z$ axis is unconstrained.

The target position is defined as a `tf2::Vector3` object, where the positions on the three axes are set to $(x, y, z) = (0.6, 0.0, 0.08)$. The orientation goals are defined as `double` variables: the target angle on the $x$ axis, Roll, is set to $-3.1416$ radians, which corresponds to $-180°$, and the target angle on the $y$ axis, Pitch, is set to 0 radians. This particular orientation allows the end effector to reach a potential object from above with an inclination suitable for a safe grasp.
Since it is not possible to isolate specific axes directly in a quaternion representation, a special technique has to be employed to enforce partial constraints.
Leaving the $z$ axis unconstrained introduces redundancy in the system. A complete pose in 3D space is defined by 6 parameters (3 for position and 3 for orientation). In this case, the 3 parameters for position are specified, while only 2 for orientation are defined. Since the robot is capable of controlling 6 degrees of freedom, having a task defined by only 5 constraints means there is an extra degree of freedom available.
In this way it is possible to optimize other secondary criteria to improve the overall performance better. Criteria like "avoid joints limits" and "minimizing joint motion" are used in this part.
Lastly, the final configuration must be collision-free, guaranteeing both that there are not intersections between the robot and the walls or with itself.

## 7.2 Scenario

The robot operates from a fixed base, performing precision tasks within a constrained workspace.
To introduce spatial restrictions, two static walls are added to the environment, positioned symmetrically in front of the robot, as shown in Figure 7.1. They define the available workspace and limit the range of feasible motions.
The two walls have the same dimensions, which are:

- length on $x$ axis: $l = 0.6m$
- thickness on $y$ axis: $t = 0.1m$
- height on $z$ axis: $h = 0.2m$

They are positioned symmetrically in front of the robot:

- the center of the left wall is positioned in $(x, y, z) = (0.6, -0.2, 0.1)$,
- the center of the right wall is positioned in $(x, y, z) = (0.6, 0.2, 0.1)$.

The available space for solutions, due to constraints, is restricted to a thickness of 0.2 meters on the $y$ axis.
The target position is fixed and located within this confined area.

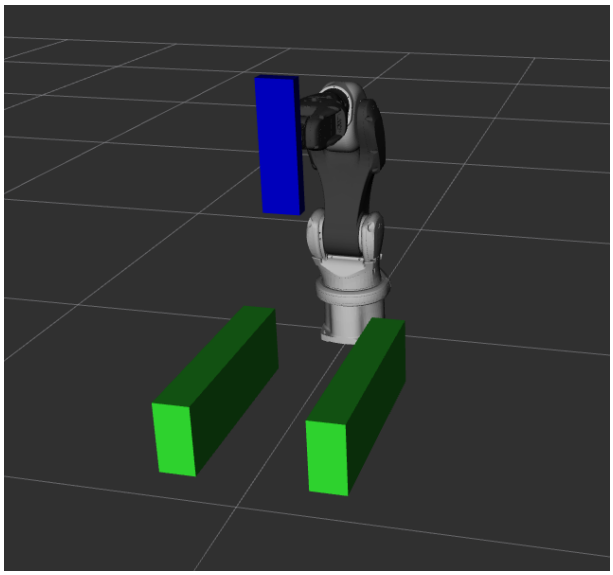

**Figure 7.1:** Symmetric walls in the scenario

The environment in which the robotic arm operates needs to be modeled correctly to ensure safe and feasible configurations. The Planning Scene Interface facilitates the dynamic addiction of collision objects, allowing the simulation of a structured environment. A `PlanningSceneInterface` object is created, which provides a mechanism to modify collision objects from the environment. It makes interaction with the scene possible, but it

does not allow direct access to collision detection computations. Then, a `PlanningScene` object is initiated with the robot model. It enables low-level access to collision checking and constraint evaluation.

## 7.3 Initialization

First, to enable motion planning and allow inverse kinematics computations, MoveIt! is initialized, and the robot model is configured.
In MoveIt! a set of joints and links that are controlled together as a unit, is called Planning Group. It defines the subset of the robot that will be considered for inverse kinematics and afterwards for motion planning.
The simplest user interface is through the `MoveGroupInterface` class, which provides a way for setting goals, managing objects and controlling the motion. This interface communicates over ROS topics, services, and actions to the MoveGroup Node.
The RobotModel and RobotState classes are the core classes that give access to a robot's kinematics. The robot's kinematic model is loaded from the ROS parameter server, where it is stored as a Unified Robot Description Format (URDF) representation. The `RobotModel` class contains the relationships among all links and joints, including their joint limit properties as loaded from the URDF.
Then, the `RobotState` object is initialized. It contains information about the robot at a certain point in time, storing vectors of joint positions and optionally velocities and accelerations. This object can be used to obtain kinematic solutions, evaluate collision constraints, and setting target configurations.
Finally, the `JointModelGroup` defines the set of joints involved in motion planning, providing access to the joint parameters and constraints, allowing for inverse kinematics computations.

## 7.4 Goals definition

As mentioned before, the particular target configuration introduces redundancy in the system, which allows for additional optimization objectives. The goals are set using goal classes, each designed to contain a specific objective. These classes evaluate errors which are then incorporated into the total cost function, so that the solution converges towards the desired location. The goals are defined and added to the `bioik_options` object, which is responsible for configuring the inverse kinematics query. Four different goal classes are used: one for the position goal, one for the partial orientation goal, one to avoid joint limits and one to keep the displacement minimal.

### 7.4.1 Position goal

For the position target, BioIK provides a goal class called `PositionGoal`, that evaluates the squared error between the current position and the target position.

First, the position goal is defined using a `PositionGoal` object. The method `setLinkName` is called to specify that the goal applies to the end effector, then the method `setPosition` is called to assign the target position coordinates to the goal. The weight for this goal is assigned to 1, since, together with the orientation, it is the main one in the problem.

### 7.4.2 Orientation goal

For the orientation target, BioIK provides a `OrientationGoal` class, which works with full quaternion representations. However, because the task requires fixing the orientation on only two axes leaving the $z$ axis free, this class is not efficient.
To solve this problem, a custom orientation goal class `PartialOrientationGoal` was developed. The class does not use quaternions to handle rotation, but it uses the RPY angles convention. By doing so, it is possible to handle the orientations on the three axes separately, being each one independent from the others. A target Roll (angle on the $x$ axis) and a target Pitch (angle on the $y$ axis) are set, while the Yaw (angle on the $z$ axis) is left unconstrained. For any given joint configuration, the class calculates the difference between the target Roll and the current Roll, and the difference between the target Pitch and the current Pitch. These two values are then squared and summed together, producing an error metric which quantifies how close the current solution is from the target one. The error is then returned to the algorithm for the optimization process.

**Listing 7.1:** PartialOrientationGoal class

```
class PartialOrientationGoal extends bio_ik::LinkGoalBase

declare roll_ as real
declare pitch_ as real

DefaultConstructor: PartialOrientationGoal()
    roll_ ← 0
    pitch_ ← 0
END DefaultConstructor

Constructor: PartialOrientationGoal(link_name, roll, pitch, weight
   ← 1.0)
    call ParentConstructor LinkGoalBase(link_name, weight)
        roll_ ← roll
        pitch_ ← pitch
END Constructor

    // Getters
    function getRoll()
        return roll_
    END function

    function getPitch()
```

```
23          return pitch_
24      END function
25
26      // Setters
27      procedure setRoll(roll)
28          roll_ ← roll
29      END procedure
30
31      procedure setPitch(pitch)
32          pitch_ ← pitch
33      END procedure
34
35      // Evaluate the goal
36      function evaluate(context)
37          // Get current orientation from context
38          current_orientation ← context.getLinkFrame().getOrientation
        ()
39          // Convert quaternion to RPY angles
40          matrix ← convert current_orientation to Matrix3x3
41          current_roll, current_pitch, current_yaw ← matrix.getRPY()
42          // Calculate difference of roll and pitch
43          roll_difference_squared ← (current_roll - roll_)^2
44          pitch_difference_squared ← (current_pitch - pitch_)^2
45          return roll_difference_squared + pitch_difference_squared
46      END function
```

The class inherits from a base goal class `LinkGoalBase`, which defines goals related to a specific robot link in the context. It provides a mechanism to associate a goal with a particular link, assigning it a weight, and registering it within the goal evaluation framework. The class has two member variables: `roll_` and `pitch_`. They store, respectively, the target Roll and the target Pitch. The default constructor initializes `roll_` and `pitch_` to 0, representing no rotation in either axis. The parameterized constructor allows initialization with specific values for the link name, Roll, Pitch, and an optional weight which determines the importance of this goal in the overall IK problem. As mentioned before in Subchapter 7.4.1, since the orientation target has higher priority over the others in the optimization, the goal weight is set to 1.

The setter functions `getRoll()` and `getPitch()` are used to get the values of the target Roll and Pitch, while the getter functions `setRoll()` and `setPitch()` can be used to update the target angles if needed.

Information is exchanged via a `GoalContext` object. The difference between the current angles and the target angles is calculated by the evaluate function. First, the current orientation in form of quaternion is recalled using `context.getLinkFrame().getOrientation()`. Then it is converted into RPY angles, separating the Roll, Pitch and Yaw components. Then the difference between the current Roll and the target Roll, and the difference

between the current Pitch and target Pitch are calculated, and squared to emphasize larger deviations. They are then summed and returned as the evaluation value which is used during the IK optimization process to search for the optimal solution.

### 7.4.3 Avoid joint limits goal

The joints have mechanical limits that is better to avoid to reduce wear and tear and to extend the lifespan of the hardware. Also, when the joints operate near their limits, motion can become unstable, or certain configurations may lead to singularities, and consequently it is difficult to ensure a smooth movement. Better reactivity and flexibility in adjustment are allowed if the joints are kept within a safe range. Therefore, this goal is useful to make sure that the robot's movements are efficient and stable.
BioIK employs a goal class called `AvoidJointLimitsGoal`, that is used for this purpose, by trying to keep the joint values in the middle half of the joint limits, and penalizing the values close to the joint limits.
The weight of the goal is assigned to 0.8, indicating its importance relative to other goals. It is slightly less significant than the position and orientation constraints, which are the primary goals for the configuration.

### 7.4.4 Minimal displacement goal

It is important to minimize unnecessary movements, making the transition to the target pose smoother and more efficient. By doing so, the robot avoids sudden or excessive joint changes, leading to a more natural motion and reducing the mechanical wear. Also, smaller joint changes result in lower energy consumption. This optimization is very useful in manipulator tasks as well as repetitive actions, where precise and controlled movements are needed. BioIK provides a goal class, called `MinimalDisplacementGoal`, which tries to find a solution that is as close as possible to the starting joint configuration.
The weight is set to 0.8, for the same reasons mentioned before to avoid joint limits goal, as in Subchapter 7.4.3.
Once all goals are defined, they are added to the goal list inside `bioik_options` using the `goals.emplace_back()` function. This action wraps all the goals together, ready for the inverse kinematics query, which will be used later to calculate a valid joint configuration that satisfies the specified constraints.

## 7.5 Inverse kinematics

The function used to find the joint configuration, is `setFromIK`, a standard function in MoveIt! that computes the inverse kinematics solution by finding a set of joint angles that allow the robot to reach a target end effector pose.

The function definition is:

**Listing 7.2:** setFromIK function definition

```
bool setFromIK(
    const JointModelGroup*  group,
    const geometry_msgs::Pose&  pose,
    double timeout = 0.0,
    const GroupStateValidityCallbackFn&
          constraint = GroupStateValidityCallbackFn(),
    const kinematics::KinematicsQueryOptions&
          options = kinematics::KinematicsQueryOptions()
);
```

where the parameters are:

- `group`: a pointer to the `JointModelGroup` that represents the kinematic chain of the robot
- `pose`: the pose that the last link in the chain needs to achieve
- `timeout`: the timeout passed to the kinematics solver on each attempt
- `constraint`: a state validity constraint to be required for IK solutions
- `options`: provides additional settings that influence how the inverse kinematics solver behaves

The function returns `true` if successful, and `false` if no valid solution is found or if the timeout is over.
To ensure a collision free solution, a callback function `collision_check_fn` is passed as a state validity constraint. The process will be better explained in details in Section 7.6.
To improve the standard MoveIt! IK solver, the BioIK framework is incorporated passing `BioIKKinematicsQueryOptions` as the option parameter to `setFromIK`. This allows `setFromIK` to use BioIK instead of the default IK solver. The `replace` flag in the options modifies the solver's handling of goal poses. By setting `replace = true` the goal pose list used by the IK methods is disabled, and the solver will focus only on the objective goals provided for the query. Moreover, the option of finding an approximate solution is present. Even if it is not enabled in this case, it could be useful for more flexibility when an exact solution is difficult to obtain, due to hard constraints or complex situations.

**Listing 7.3:** IK algorithm

```
// IK search initialization
max_attempts ← 10
success ← false

FOR attempt ← 0 TO max_attempts DO
    // Randomize starting joint positions
    robot_state ← set joints to random positions
```

```
    robot_state ← update
    planning_scene ← set current state
    // Compute IK solution
    success ← setFromIK(robot_state, joint_model_group, target_pose
    , 10.0, collision_check_fn, bioik_options)

    IF success THEN
        // Create a vector for joint values
        declare joint_values
        // Copy joint positions from robot state
        joint_values ← copy joint positions
        print joint_values
        // Calculate forward kinematics
        robot_state ← set joint values
        robot_state ← update
        // Get the end effector transform
        end_effector_transform ← get end effector transform
        // Create and update end effector pose
        declare end_effector_pose
        end_effector_pose.position ← extract from
    end_effector_transform
        end_effector_pose.orientation ← extract from
    end_effector_transform
        // Convert orientation to quaternion and RPY angles
        final_orientation_quaternion ← convert
    end_effector_orientation to quaternion
        matrix ← convert final_orientation_quaternion to Matrix3x3
        final_roll, final_pitch, final_yaw ← matrix.getRPY()
        print FK results
        // Calculate errors
        position_difference ← target_position - end_effector_pose.
    position
        roll_difference ← target_roll - final_roll
        pitch_difference ← target_pitch - final_pitch
        print final position and angles errors
        BREAK
    ELSE
        print error message
    END IF
END FOR
```

The `setFromIK` function is incorporated into a *for loop*, that allows for the search of a valid and collision free solution.

The loop executes a number of attempts, with a maximum number of attempts, `max_attempts`, that represents how many times the IK solver will retry before giving up (in this case, the maximum number of attempts is set to 10). By limiting the number

of attempts, the algorithm prevents excessive computation time and ensures that the IK solver does not run indefinitely.
On each iteration of the loop, the `setToRandomPositions` method is called to generate a new random starting configuration for the robot's joints. Thus, the joint positions are given a random value, which is within their valid ranges, providing different initial configurations for every attempt. This randomization is crucial for increasing the likelihood of finding a valid solution, as it prevents the solver from getting trapped in local minima or from starting from poor initial guesses that might lead to failure. Essentially, the randomization helps the solver to explore a wider portion of the solution space, improving the chances of success by diversifying the starting points.
Once the starting configuration is set, the robot state is updated to make sure that the planning scene has the correct starting configuration.
Then, the `setFromIK` function is called to compute the solution. It attempts to find the joint values that allow to satisfy the goals, within a time limit of 10 seconds. There are two possible outcomes: either a valid solution is found within the time limit, or it is not found before the time limit expires.
If a valid solution is not found within the time limit, the loop proceeds to the next attempt, setting as starting joint positions some new random values (always within the joint ranges). Each failure is displayed as a warning message, indicating that a solution was not found. If the failure is due to a collision, the warning message also includes the corresponding joint values and identifies the sources of collision, providing better understanding of the outcome.
Instead, if a valid solution is found, the loop exits early, since there is no need to continue the search.
The final joint values are extracted from the robot state and saved into a vector called `joint_values`. The robot state is then updated with such new values and the forward kinematics is calculated to confirm that the found joint position results indeed in the expected end effector pose. Also, a collision check is performed again, to make sure that the end effector does not collide nor violate some constraints.
To get the timing information essential for performance analysis, `ros::Time` is used. Two different time intervals are measured: the time taken for each `setFromIK` attempt and the total time taken by the whole search to find the solution.
The target end effector position, the target Roll and Pitch, the number of attempts, and the time limit for each attempt (the `setFromIK` timeout) can be adjusted depending on the specific requirements of the task. This flexibility makes the approach suitable for different kinds of robotic applications.

## 7.6 Collision avoidance

Collision checking is the process of evaluating whether an object or a system intersects with obstacle or other objects in the environment. It is widely used in robotics and simulations

to ensure that the robot avoids obstacles while executing tasks in a workspace. The process involves detecting intersections between geometric shapes, which can range from simple boxes to complex mesh models.
To ensure that the computed inverse kinematics solution results in a feasible and safe configuration, a collision-checking mechanism is applied.
The collision checking function is implemented.

**Listing 7.4:** Collision check function

```
function collisionCheckFn(planning_scene, robot_state,
    joint_model_group, joint_values)
     // Set and update robot state
     robot_state ← planning_scene.setCurrentState
     robot_state ← setJointGroupPositions(joint_model_group,
    joint_values)
     robot_state ← update
     declare collision_request
     declare collision_result
     collision_result ← clear
     collision_request.contacts ← true
     planning_scene.CheckCollision(collision_request,
    collision_result, robot_state)
     IF collision_result.collision = true THEN
          print warning message
          return false
     ELSE
          return true
     END IF
END function
```

The function takes as parameters the current planning scene, a pointer to the robot's state, the joint model group that specifies which joints to update, and an array of joint values representing the new configuration.
First, the robot state in the planning scene is set to the provided `robot_state`. The planning scene encapsulates the robot's environment, which includes the robot itself as well as any object in the world. The collision check needs to be made on the most up to date configuration, so the joint values are set to the ones provided by the `joint_values` array, and the `robot_state` is updated.
Then, two objects are initialized: `collision_request` and `collision_result`. The first one specifies the parameters for the collision check and it is used to request contact information from the planning scene, while the second one holds the outcome of the collision check, by storing whether a collision occurred, along with any relevant contact details. The `collision_result.clear()` method is called before each check to reset the `CollisionResult` object to ensure that previous results do not interfere with the active check. The collision check is performed using `planning_scene.checkCollision()`

method. It evaluates the robot's configuration against the environment and, if any part of the robot is in contact with an obstacle or another robot part, the result will be a collision. If the configuration is collision free, the function returns `true`, otherwise, it outputs a warning and returns `false`. The objects that are in collision are also displayed to have a better understanding of the situation.
Now, the `collisionCheck` function needs to be integrated with the function `setFromIK`, which uses a callback function to define additional constraints and to check the validity of the robot configuration. The function signature expects three parameters: `robot_state`, `joint_model_group` and `joint_values`. It returns `true` if the configuration is valid, `false` if it is invalid.

**Listing 7.5:** setFromIK callback function signature

```
bool callBackFn(moveit::core::RobotState* robot_state,
                const moveit::core::JointModelGroup* joint_model_group,
                const double* joint_values)
```

To match the expected signature, the callback function would need to be defined using just those three parameters, without allowing for extra arguments. However, the `planning_scene` parameter is needed to perform collision checking. To overcome this problem, a lambda function is used to capture the needed external variable and pass it to the collision check function.

**Listing 7.6:** Lambda function used for collision check

```
const moveit::core::GroupStateValidityCallbackFn collision_check_fn =
[&planning_scene](moveit::core::RobotState* robot_state,
                  const moveit::core::JointModelGroup*
    pJoint_model_group,
                  const double* joint_values) {
      return collisionCheckFn(planning_scene, robot_state,
    pJoint_model_group, joint_values);
    };
```

The lambda function is then passed as a parameter to the `setFromIK` function. If the collision check returns a successful outcome, meaning no collisions has occurred, the `setFromIK` function returns `true`, the corresponding joint values are saved and the search for a solution stops. Instead, if the collision check indicates a failure, meaning that a collision has occurred, the `setFromIK` function returns `false`, and the search continues to find a valid solution.
Collision check is also performed after the joint values are obtained, serving as an additional validation step to ensure that the solution is collision free before finalizing the result. The outcome of the check is then printed out.

## 7.7 Motion planning

Motion planning in MoveIt! is the process of computing a sequence of robot joint or end-effector positions that move a robot from its current configuration to a target one, while satisfying a set of constraints, such as avoiding collisions, respecting joint limits, or adhering to specific orientation goals.
OMPL (Open Motion Planning Library) [38] is an open-source library that provides various algorithms for motion planning. It is designed to be a flexible and extensible framework for sampling-based motion planning and is commonly used with MoveIt! to plan robots' paths in different environments. The algorithms are based on sampling methods, useful for complex environments. Some of the most used algorithms are RRT (Rapidly-exploring Random Tree), PRM (Probabilistic Roadmap), EST (Expansive Space Trees).
During a first try, once the joint configuration is found with the IK proposed method, the robot moves to the target position using OMPL planning.
Then, to improve the trajectory, a waypoints method was implemented, breaking down the motion into intermediate steps to secure smoother, optimized movement.
This process moves the robot's end effector from a start pose to a final pose, computing intermediate poses along the way. The poses are calculated using interpolation and for each pose, the optimal joint configuration is evaluated. The robot then moves through successive poses with OMPL motion planning.
The target pose that is found by doing the forward kinematics with the optimal joint values found before, is saved as the final pose, while the starting pose of the robot is saved as the start pose.
The interpolation of the intermediate points is handled in a different way for position and orientation:

- Position: a linear interpolation approach (LERP) is used for positional displacement. The linear interpolation calculates one intermediate position between two points using the *step* parameter, which represents a fraction of the way between the start and final values. Its range varies between 0 and 1, as follows:
  - $step = 0$: the interpolation is at the starting value,
  - $step = 1$: the interpolation is at the final value,
  - $0 < step < 1$: depending on the value of the step, the interpolation is somewhere in between the starting and final values.

  At each iteration, the step is incremented by 0.1, until a maximum value of 1.
  Indicating with $P_{intermediate}$ the intermediate position, $P$ the current position, $P_{final}$ the final position, and $\alpha$ the step, the linear interpolation formula is:

$$P_{intermediate} = P + \alpha(P_{final} - P) \tag{7.1}$$

- Orientation: a spherical linear interpolation approach (SLERP) is used for orientation

displacement. The spherical interpolation calculates one intermediate orientation between two quaternions, always using the *step* parameter as described in the case of linear interpolation.
Indicating with $\theta = cos^{-1}(Q \cdot Q_{final})$ the angle between the quaternions, $Q_{intermediate}$ the intermediate quaternion, $Q$ the current quaternion, $Q_{final}$ the final position, and $\alpha$ the step, the spherical interpolation formula is:

$$Q_{intermediate} = \frac{sin((1-\alpha)\theta)}{sin(\theta)}Q + \frac{sin(\alpha\theta)}{sin(\theta)}Q_{final} \tag{7.2}$$

First, a position threshold, orientation threshold and time limit are chosen. The algorithm loop runs until either it exceeds the time threshold, or until the final pose is reached inside the acceptable threshold. The time is tracked using `ros::Time::now() - initial_time`. The starting pose of the end effector is saved as the current pose, and the pose computed before with the IK algorithm is set as the final pose to reach. At the beginning of each iteration, the robot position and orientation are extracted from the current pose, and stored, respectively, as `Eigen::Vector3d`, and `Eigen::Quaterniond`. Then, the difference between the current pose and final pose is computed. For the position, the squared Euclidean distance between the current and final positions is computed, while for the orientation, the difference is calculated using the dot product of quaternions, ensuring the shortest path is chosen for rotation. If both the position and orientation differences are lower than the defined thresholds, the goal is reached and the loop exits. Instead, if the final pose is not reached yet, an intermediate pose is calculated: using a linear interpolation for the position, and a spherical one for the orientation. The step size, $\alpha$, is initialized with a value of 0 and increased by 0.1 at each iteration, ensuring a gradual transition toward the target. For each intermediate pose, an inverse kinematics solution is computed. The goals that are used to optimize the configurations are similar to the ones previously used to find the final pose, with one difference for the orientation goal. This time, since the final orientation is fully defined as a quaternion, the orientation class provided by BioIK is used. Therefore, the four optimization goals that are used are:

- Position goal: to reach the interpolated position.
- Orientation goal: to reach the interpolated orientation.
- Avoid joint limits goal: to stay within the center range of the joints.
- Minimal displacement goal: to decrease the joint movements between poses, which is useful to generate smooth transitions.

Then, the inverse kinematic solution is computed using the strategy described before. A for loop attempts to find a solution within a predetermined number of tries: for each try the robot's current joint values are recalled and set as the initial state, then the planning scene is updated to reflect the robot's state, and finally the IK solver `setFromIK` is used with collision checking to compute a feasible solution. When a valid IK solution is found,

the corresponding joint values are extracted, forward kinematics is used to get the exact obtained intermediate pose, and an additional collision check ensures the solution is safe. Afterward, the target joint values are set in MoveIt!, a motion plan is generated and, if successful, the robot executes the motion. Finally, the current pose is updated to the intermediate pose, and the process repeats until the final pose is reached or until the time limit is exceeded.

**Listing 7.7:** Optimized motion pseudocode

```
position_threshold ← 0.0001
orientation_threshold ← 0.001
time_threshold ← 60.0
step ← 0.1
marker_id ← 0
current_pose ← start_pose
final_pose ← final_end_effector_pose
final_position ← final_end_effector_pose.position
final_orientation ← final_end_effector_pose.orientation
// Time tracking
initial_time ← current_time
WHILE current_time() - initial_time < time_threshold DO
    // Extract current position and orientation from current pose
    current_position ← current_pose.position
    current_orientation ← current_pose.orientation

    visualizeMarker in current_position
    marker_id ← marker_id + 1

    // Calculate position and orientation differences
    position_diff ← (final_position - current_position).squaredNorm
    ()
    dot_prod ← final_orientation.dot(current_orientation)
    IF dot_prod < 0.0 THEN
        final_orientation ← -final_orientation
        dot_prod ← -dot_prod
    END IF
    dot_prod ← max(-1, min(1,dot_prod)) // clamp in range [-1;1]
    orientation_diff ← 2 * acos(dot_prod)

    IF (position_diff < position_threshold AND orientation_diff <
    orientation_threshold) THEN
        BREAK
    END IF

    // Compute intermediate pose
    declare intermediate_pose
    intermediate_pose.position ← current_pose.position + (
```

```
    final_pose.position - current_pose.position) * step
37   intermediate
38   intermediate_pose.orientation ← current_orientation.slerp(step,
     final_orientation)
39
40   step ← step + 0.1
41   IF step > 1 THEN
42       step ← 1
43   END IF
44
45   // Define goals
46   // Position goal
47   position_goal ← PositionGoal()
48   position_goal.setLinkName(end_effector)
49   position_goal.setPosition(intermediate_pose.position)
50   // Orientation goal
51   orientation_goal ← OrientationGoal()
52   orientation_goal.setLinkName(end_effector)
53   orientation_goal.setOrientation(intermediate_pose.orientation)
54   // Avoid joint limits goal
55   avoid_joint_limits_goal ← AvoidJointLimits()
56   avoid_joint_limits_goal.weight ← 0.8
57   // Minimal displacement goal
58   min_displ_goal ← MinimalDisplacementGoal()
59   min_displ_goal.weight ← 0.8
60   bioik_options.goal ← clear
61   bioik_options.goal ← add(position_goal, orientation_goal,
    avoid_joint_limits_goal, min_displ_goal)
62
63   // Start IK search
64   max_attempts ← 10
65   success ← false
66   joint_values ← []
67   FOR attemt ← 0 TO max_attempts DO
68       current_joint_values ← get current joint positions from
    robot_state
69       robot_state ← set current_joint_values
70       robot_state ← update
71       planning_scene ← set current robot state
72       // Compute IK solution
73       success ← setFromIK(joint_model_group, target_pose, 10.0,
    collision_check_fn, bioik_options)
74       IF success = true THEN
75           joint_values ← copy joint positions
76           // Forward kinematics
77           robot_state ← set joint values
78           robot_state ← update
```

```
79              intermed_transf ← get global link transform
80              intermediate_pose.position ← get position from
    intermed_transf
81              intermediate_pose.orientation ← get orientation from
    intermed_transf
82
83              declare collision_request
84              declare collision_result
85              collision_result ← clear
86              planning_scene ← check collision
87
88              IF collision_result.collision = true THEN
89                  print error message
90              END IF
91
92              print forward kinematics results
93              BREAK
94          ELSE
95              print error message
96          END IF
97      END FOR
98
99      target_joint_values ← joint_values
100     plan ← move_group_interface.plan
101     motion_success ← plan motion
102     IF motion_success = true THEN
103         print success message
104         plan ← execute motion
105     ELSE
106         print error message
107     END IF
108
109     current_pose ← intermediate pose
110 END WHILE
```

The optimized path is marked using markers in the intermediate positions. Markers are used for visualization purposes, allowing users to display objects, points, or trajectories in RViz. They are part of the `visualization_msgs::Marker` message type and can represent various shapes, such as spheres, arrows, or cubes. In this case, spherical markers are used. A marker is published in every intermediate position that the robot reaches; the marker time limit is set to an unlimited time, to ensure that at the end of the process it is possible to visualize the whole path.

Figure 7.2 shows one possible optimized path, created from an asymmetric starting position to the final position inside the walls. It can be observed that the distance between the points decreases as the end effector approaches the target pose, until it converges on it.

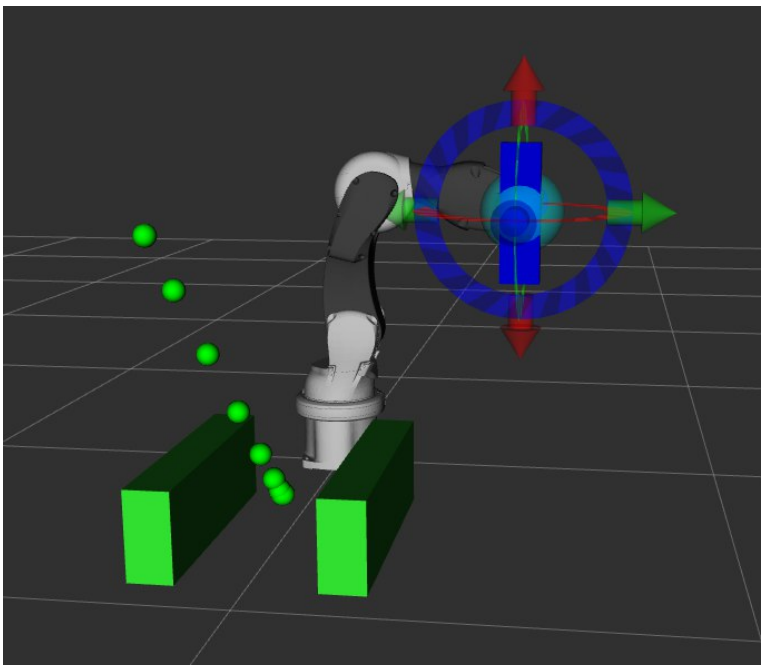

**Figure 7.2:** Intermediate poses of the optimized path

# Chapter 8

# Testing and results

In this chapter the results obtained by the IK algorithm are presented and analyzed. The algorithm performances are tested first in a simulated environment, then on a real robot. The metrics that are used for the evaluation are:

- Average final pose error, to determine the accuracy of the solution.
- Average computation time of the solution, to determine the algorithm computational efficiency.
- Success rate, to determine the percentage of successful attempts in finding a solution.

Finally, the effectiveness of the addition of the secondary goal is verified with different algorithm tries.

## 8.1 Simulation results

### 8.1.1 Primary goals

First, the algorithm is tested in a ROS simulation environment. A target pose composed by a full target position and two constrained axes, is chosen inside the walls, and the algorithm is executed 100 times for the selected pose. For each execution the position and orientation errors, and the computation time are collected.
The mean of these results is calculated to have a general performance estimation, while the standard deviation is calculated for algorithm stability and outlier detection.

#### Position error

The position error is the difference between the target position and the actual final position reached by the end effector. It measures how accurate the solution is, and it is defined as the Euclidean distance between the two positions. Indicating the target position as $p_T = (x_T, y_T, z_T)$ and the achieved position as $p = (x, y, z)$, the position error $e_p$ is calculated as:

$$e_p = \|p_T - p\| = \sqrt{(x_T - x)^2 + (y_T - y)^2 + (z_T - z)^2} \tag{8.1}$$

The average position error is $\bar{e_p} = 7.20997 \cdot 10^{-6}m$, and the standard deviation is $\sigma_{e_p} = 3.72674 \cdot 10^{-6}m$.
The results demonstrate that the algorithm achieves highly precise and consistent positioning, with errors in the order of micrometers. This level of accuracy is sufficient for most robotic applications, confirming that the inverse kinematics solver is both reliable and effective.

### Orientation error

Since the orientation is constrained on two axes, the full orientation metrics (such as quaternion distance) can not be used. Therefore, only the errors on the constrained axes are compared. The difference between the target Roll $\psi_T$ and the actual Roll $\psi$, and the difference between the target Pitch $\theta_T$ and the actual Pitch $\theta$ are calculated.
The Roll error, $e_\psi$, is calculated as:

$$e_\psi = |\psi_T - \psi| \tag{8.2}$$

The Pitch error, $e_\theta$, is calculated as:

$$e_\theta = |\theta_T - \theta| \tag{8.3}$$

The average Roll error is $\bar{e_\psi} = 8.03388 \cdot 10^{-6}rad$, and the standard deviation is $\sigma_{e_\psi} = 8.23785 \cdot 10^{-7}rad$, while the average Pitch error is $\bar{e_\theta} = 1.75419 \cdot 10^{-6}rad$, and the standard deviation is $\sigma_{e_\theta} = 1.35452 \cdot 10^{-6}rad$. These results demonstrate that the algorithm achieves high precision and stability in finding the target angles, with errors in the micro-radian range. Such small errors are negligible in most robotic applications, meaning the system successfully maintains orientation accuracy within an excellent tolerance.
Moreover, it is observed that the Yaw $\phi$ angle changes between different tries, with a range that goes from $-3.1rad$ to $2.9rad$, meaning that it is actually unconstrained.

### Computation time

The average computation time is $0.08150914s$ with a standard deviation of $\sigma_t = 0.040219975s$. This variability is due to the algorithm's use of multiple attempts to find a solution, with each attempt having an average computation time of $0.04s$. On average, the algorithm requires 2.2 attempts per execution, which contributes to the observed variability in computation time.
The computation time of $0.08s$ is generally considered efficient, providing a good balance between accuracy and speed. This time is suitable for most applications, ensuring that the system can generate solutions in a reasonable time frame while maintaining effective performance.

#### Success rate

To evaluate the success rate of the algorithm, 500 random target points were generated within the constrained area formed by the two walls, all located in the robot's reachable zones. For each pose, the algorithm was executed to compute a valid IK solution. The results of the attempts were visualized using markers: the poses where the algorithm was successful were colored in green, while poses where the algorithm failed, were marked in red.
The algorithm achieved a success rate of 98.9%, which demonstrates high success rate in solving the inverse kinematics.

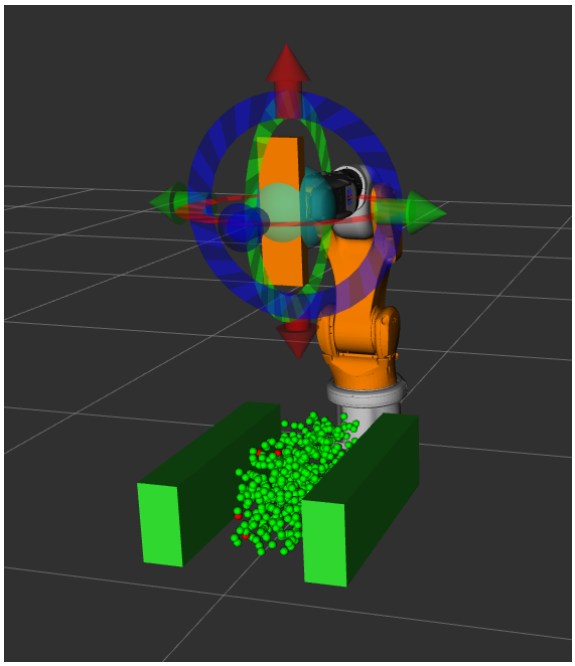

**Figure 8.1:** Testing with 500 markers

Hence, the results show that the algorithm is able to solve the inverse kinematic problem with good accuracy in a relatively short time, confirming that the $z$ axis remains effectively unconstrained.

### 8.1.2 Secondary goals

Now, the achievement of the secondary goals that are added to the primary ones is evaluated. First, only the primary goals are activated, leaving both the second ones deactivated, and the algorithm is tested over five runs. Then, the first secondary goal is added, and another five runs are performed. After that, the first secondary goal is removed, and the second secondary goal is added, to compare their individual effects. In each of these stages, the joint values are recorded and used for analysis.

### Avoid joint limits goal

This goal has the purpose to keep the joint values far from the joint limits. The joint limits are plotted together with the joint values. The situation of each joint can be analyzed in a separate line graph. On the horizontal axis of the graph the test runs are plotted, while on the vertical axis the joint values are plotted. It is possible to observe an orange line that represents the minimum limit, and a blue line that represents the maximum limit. The joint values found without the secondary goal, are plotted in green, while the joint values found with the addition of the secondary goal, are plotted in brown.

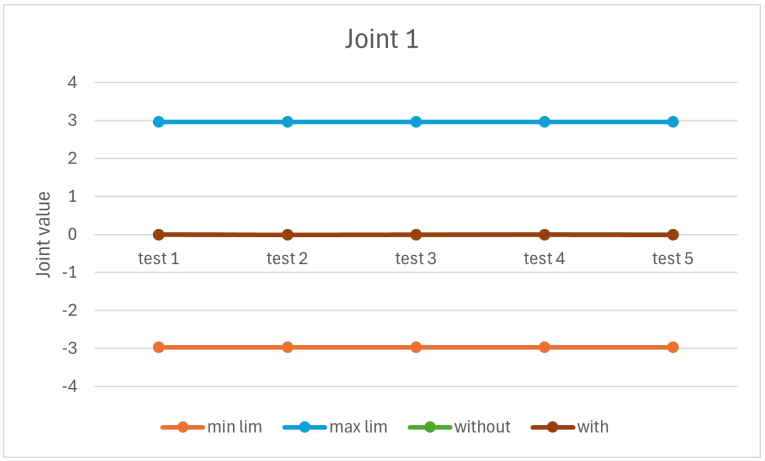


**Figure 8.2:** Comparison of the avoidance joint limits goal on joint 1

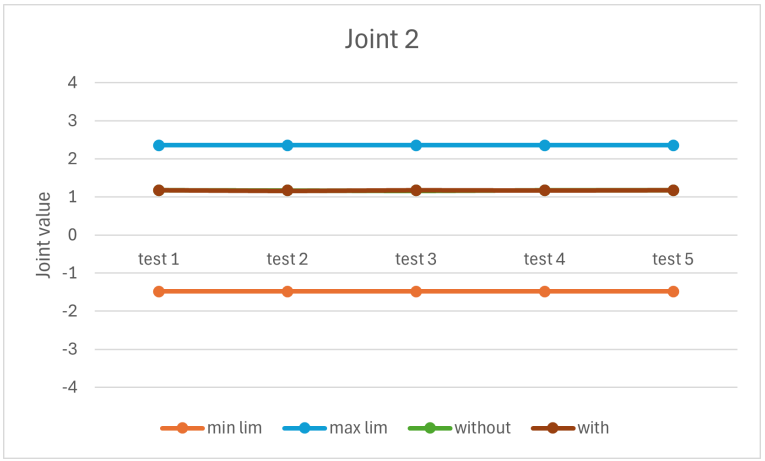


**Figure 8.3:** Comparison of the avoidance joint limits goal on joint 2

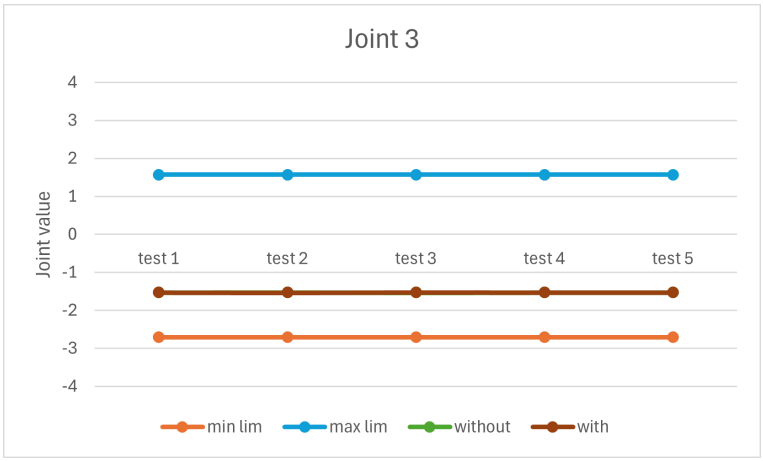


**Figure 8.4:** Comparison of the avoidance joint limits goal on joint 3

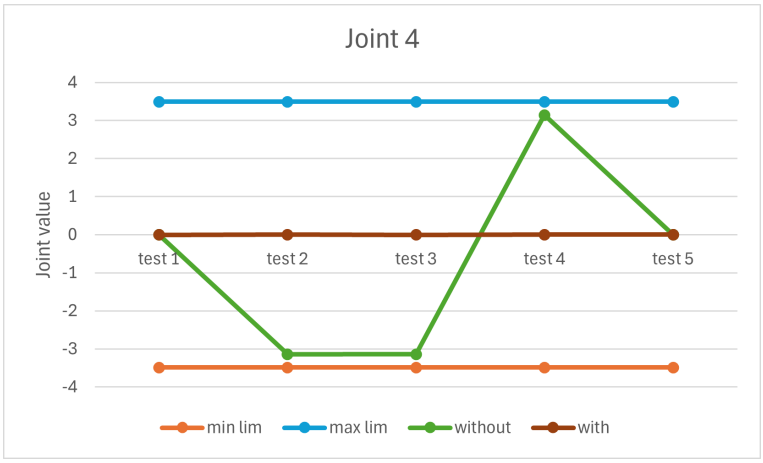


**Figure 8.5:** Comparison of the avoidance joint limits goal on joint 4

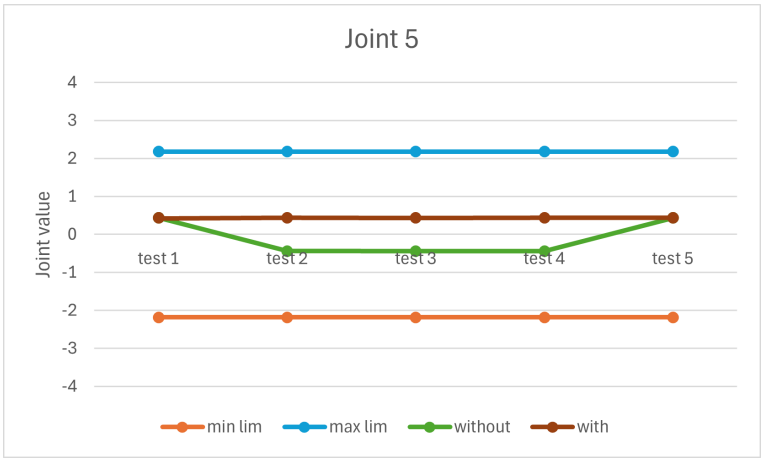


**Figure 8.6:** Comparison of the avoidance joint limits goal on joint 5

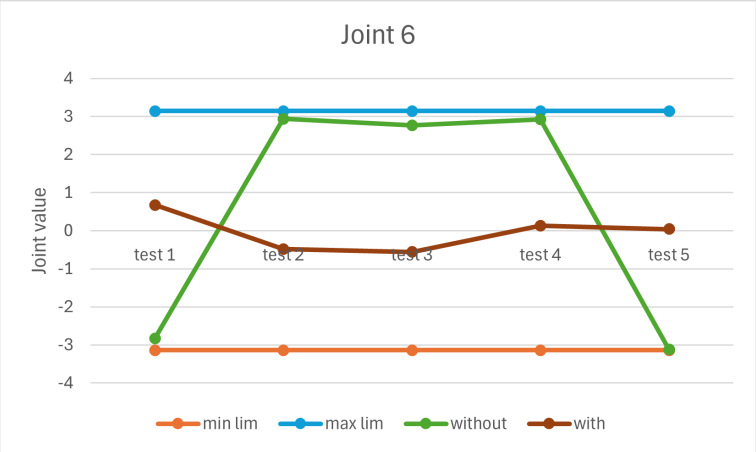


**Figure 8.7:** Comparison of the avoidance joint limits goal on joint 6

In joints number 1, 2 and 3, it is not possible to observe a significant difference, since in both cases the joints are in the middle range of their limits. In joint number 5, in the tests number 2, 3 and 4, it is possible to observe a slightly worsening of the performance in the case without the secondary goal. The most significant difference can be observed in joints 4 and 6. In the run without secondary goal, the joints are very close to the limits, while when the secondary goal is added, the joint values are kept constant inside the joint middle range.
Hence, from the obtained results, it is possible to observe that the first three joints are not affected by this secondary goal, while the last three benefit from it. This happens because the optimization of the primary goal (pose goal) occurs by exploiting the arbitrariness of one degree of freedom in the orientation, which is in fact determined by the last three joints.

### Minimal joint displacement goal

This goal has the purpose to minimize the joint displacement between configurations. The displacement is calculated as the difference between the final joint values and the starting joint values. One bar chart graph is built for each joint. On the horizontal axis the test runs are plotted and on the vertical axis the displacement is plotted. The displacements of the tries that are run without the addition of the secondary goal are plotted in blue, while the ones of the tries that are run with the secondary goal are plotted in orange.

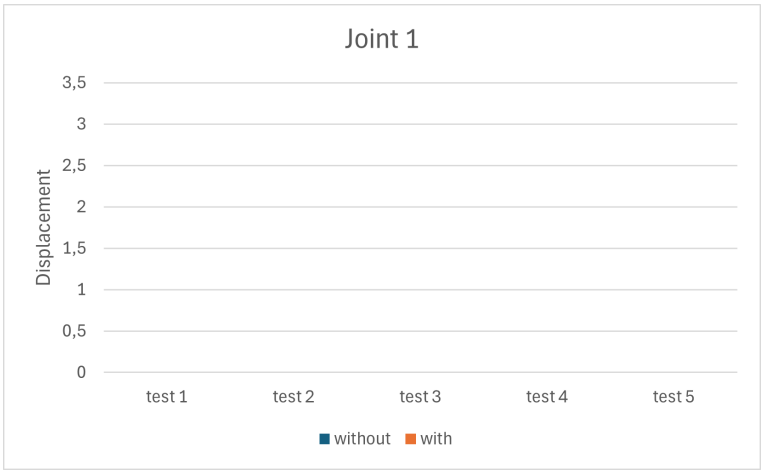


**Figure 8.8:** Comparison of the minimal joint displacement goal on joint 1

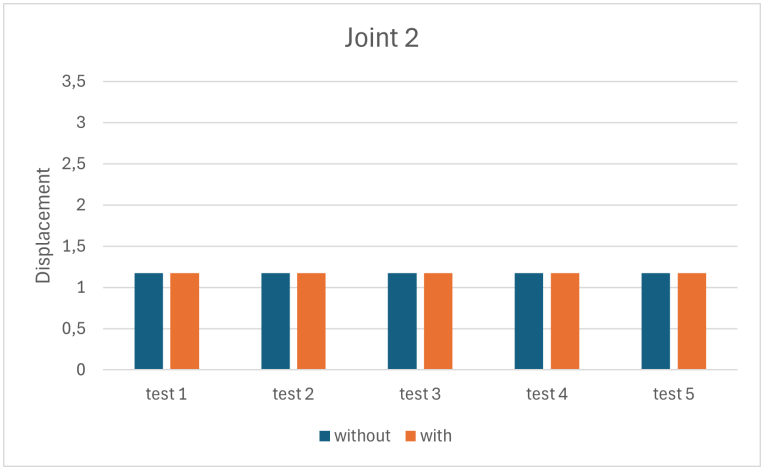


**Figure 8.9:** Comparison of the minimal joint displacement goal on joint 2

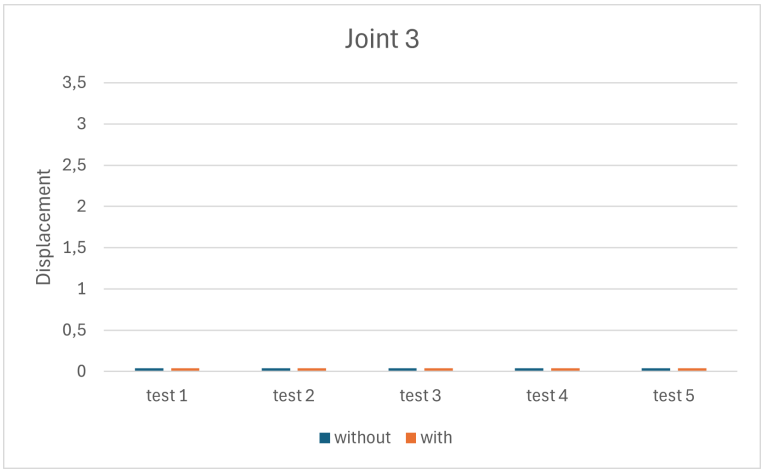


**Figure 8.10:** Comparison of the minimal joint displacement goal on joint 3

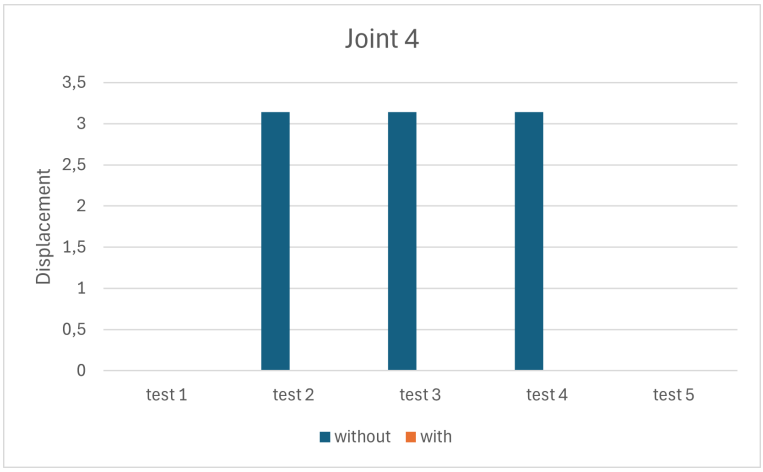


**Figure 8.11:** Comparison of the minimal joint displacement goal on joint 4

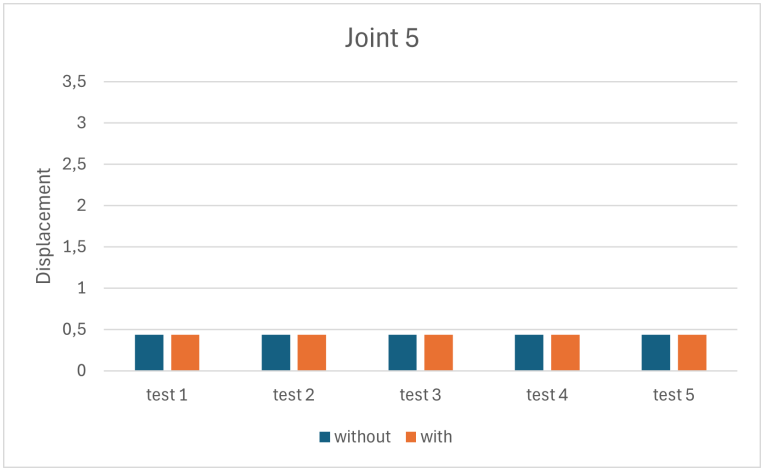


**Figure 8.12:** Comparison of the minimal joint displacement goal on joint 5

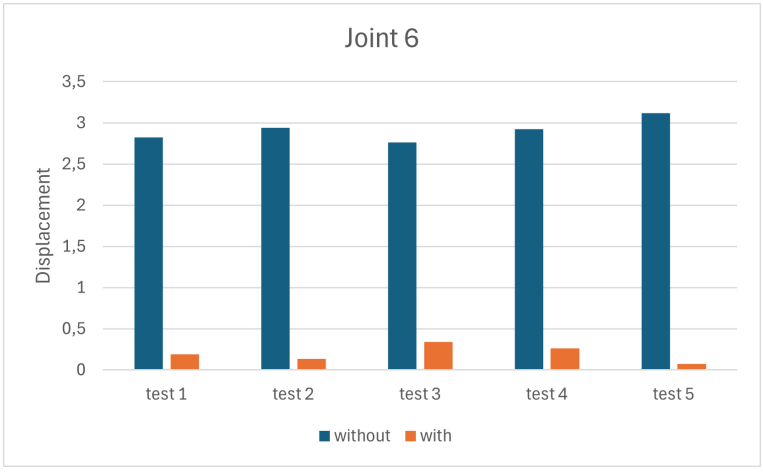


**Figure 8.13:** Comparison of the minimal joint displacement goal on joint 6

In joints number 1 and 3 no displacement is observed in both cases. In joints number 2 and 5 is present a small displacement that is constant for all the runs in both cases. The improvement of the solution can be found in joints number 4 and 6. In both joints it is possible to observe a large joint displacement when the secondary goal is not added. The results are improved when the secondary goal is added because the displacement decreases significantly, becoming zero or very small.

## 8.2 Real robot results

The algorithm is then tested on a real robotic manipulator, specifically the Racer3-COBOT. This robot shares the same joint structure as the Racer5, but has shorter link lengths. The end effector is a gripper, with a rectangular shaped object attached to it to simulate the asymmetric end effector used in the algorithm. The two walls are simulated using two boxes.
First, a driver ROS for robot Comau is configured and a connection TCP/IP is established with the physical robot. A test workspace is configured and the values of the joints of the robot are recorded. Then, a simulation is run in RViz, where the robot is moved and tested, with the results observed in real time.

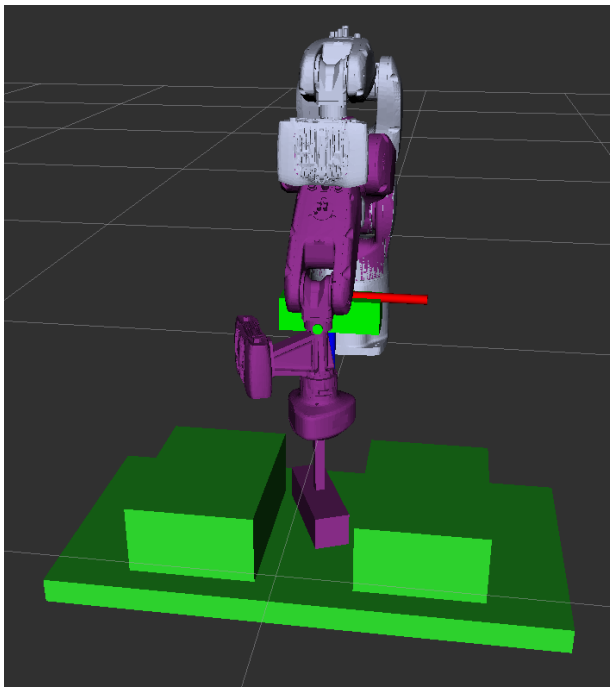

**Figure 8.14:** RViz simulation of the Racer3-COBOT

To test the inverse kinematic algorithm, first, the end effector of the robot is moved to the desired final position, which is then saved as target one. Also, the Roll and Pitch angles are extracted from the final pose and used as target angles. The boxes are placed on the table, and their exact position is accurately represented in the simulation. The robot is then moved back to its starting pose. Afterwards, the algorithm is used to find the

optimized joint configuration needed to reach the final goal pose. A client node sends the final joint values as a goal through an action to the server, which calls a callback function. This function receives the joint goal vector as parameter and sets the joints to these values. It opens a thread in parallel where the planning of the trajectory is done, and another parallel thread where the trajectory is finally executed.
The end effector successfully reaches the desired pose with an optimized configuration while avoiding collisions.


**Figure 8.15:** Real test using the Racer3-COBOT

# Chapter 9

# Conclusions

This thesis presents a solution to the inverse kinematic problem for a redundant robotic manipulator, whose redundancy is introduced by its working conditions. The goal is to reach a target position, imposing orientation constraints on the $x$ and $y$ axes, while leaving the $z$ axis unconstrained. To solve this problem, the BioIK package solver was studied, adapted and integrated within the ROS framework. A customized partial orientation goal class was specially developed to handle the required orientation goal, and other already existing goal classes were used to fulfill the position goal and enhance the overall performance. The *setFromIK* function was used and improved to find the final collision free joint configuration. The algorithm was implemented using the MoveIt! library, which provided the planning environment and tools for simulating and controlling the robot. First, it was tested in a simulation environment, and then on a real robotic manipulator.
The results show that the algorithm is able to find an acceptable solution with a good balance between accuracy and computation time.
Furthermore, this approach was also used to improve the end effector path using interpolation points. For each one, an optimized joint configuration was generated, allowing the robot to follow these configurations, making its movement smooth and efficient.

## Future work

Despite the positive outcome, improvement is always possible. It would be useful to reduce the number of attempts the algorithm takes to find a solution, which would decrease the computation time. Another improvement would be to add dynamic constraints, since they were not considered in this work, to make the algorithm more suitable for real world applications. Finally, the scenario could be made more complex by introducing dynamic obstacles or increasing the interaction with the surrounding environment.

## Other applications

There are several real world applications that do not require a fully constrained orientation, but only constraints on specific axes. For example, in tasks such as inserting an object into a slot, polishing or welding, or handling tools that require rotation flexibility, this kind of orientation target could be needed. Moreover, this approach could be combined with the interpolation of the waypoints to guarantee a smooth and optimized path for these applications.

# Bibliography


[1] Forward and inverse kinematics. http://compas.dev/compas_fab/0.28.0/examples/03_backends_ros/03_forward_and_inverse_kinematics.html, 2021.

[2] L. Villani B. Siciliano, L. Sciavicco and G. Oriolo. *Robotics, Modelling, Planning and Control*. Springer, 2009.

[3] C. Sivakumar. Robotics kinematics and dynamics. Slides for the course Robotics, 2021. BSA Crescent Institute of Science and Technology, Department of Mechanical Engineering.

[4] Alessandro Rizzo. Kinematics of manipulators. Slides for the course Robotics, 2021. Politecnico di Torino.

[5] Flavio Ferraz. A comparative study of the accuracy between two computeraided surgical simulation methods in virtual surgical planning. 2020.

[6] Masri Ayob Fahad AL-Dhief Musatafa Abbas Albadr, Sabrina Tiun. Genetic algorithm based on natural selection theory for optimization problems. 2020.

[7] Kun-Huang Kuo Chao-Hsing Hsu, Wen-Jye Shyr. Optimizing multiple interference cancellations of linear phase array based on particle swarm optimization. 2010.

[8] Wilhelmus A. M. Van Noije Tiago Oliveira Weber. Design of analog integrated circuits using simulated annealing/quenching with crossovers and particle swarm optimization. 2012.

[9] F.M. Rico. A concise introduction to robot programming with ros2. 2022.

[10] Daniel Serrano. Introduction to ros – robot operating system –. 2019.

[11] Ros 2 documentation. https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html, 2025.

[12] Ros 2 documentation. https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html, 2025.

[13] Hands-on introduction to robot operating system(ros). `https://trojrobert.github.io/hands-on-introdution-to-robot-operating-system(ros)/`, 2020.

[14] Comau. `https://www.comau.com/it/competencies/products-solutions/robot-team/racer-5-0-80/`, 2025.

[15] Alex Owen-Hill. Robot singularities: What are they and how to beat them. `https://robodk.com/blog/robot-singularities/`, 2022.

[16] D. Rose. Rotations in three-dimensions: Euler angles and rotation matrices. 2015.

[17] Ahmed Mohamed Zaki-Ahmed Mohamed Zaki El-Sayed M. El-kenawy Abdulrahman Abdullah Farag, Ziad Mohammed Ali. Exploring optimization algorithms: A review of methods and applications. 2024.

[18] Daniel Marthaler. An overview of mathematical methods for numerical optimization. 2013.

[19] Robertas Damasevicius. Patterns in heuristic optimization algorithms: A comprehensive analysis. 2025.

[20] Ricardo Gustavo Rodríguez-Canizo Javier Alexis Abdor-Sierra, Emmanuel Alejandro Merchan-Cruz. A comparative analysis of metaheuristic algorithms for solving the inverse kinematics of robot manipulators. 2022.

[21] Donald Griersonb Emad Elbeltagia, Tarek Hegazyb. Comparison among five evolutionary-based optimization algorithms. 2005.

[22] Hans-Paul Schwefel Tomas Back. An overview of evolutionary algorithms for parameter optimization. 1993.

[23] Lei Liu Dongshu Wang, Dapei Tan. Particle swarm optimization algorithm: an overview. 2017.

[24] Jyoti Jain N.K.Jain, Uma Nangia. A review of particle swarm optimization. 2018.

[25] Freisleben B. Merz P. A genetic local search approach to the quadratic assignment problem. 1999.

[26] Carlos Cotta Ferrante Neri. Memetic algorithms and memetic computing optimization: A literature review. 2011.

[27] Carla P. Gomes Bart Selman. Hill-climbing search. 2006.

[28] Peter J. M. van Laarhoven Emile H.L. Aarts, Jan H. M. Korst. Simulated annealing. 1997.

[29] Sebastian Ruder. An overview of gradient descent optimization algorithms. 2017.

[30] Wheeler Ruml Christopher Wilt, Jordan Thayer. A comparison of greedy search algorithms. 2010.

[31] Robot Operating System. Robot operating system (ros). `http://www.ros.org/`.

[32] William D. Smart Morgan Quigley, Brian Gerkey. *Programming Robots with ROS: A Practical Introduction to the Robot Operating*. O'Reilly Media, 2015.

[33] Ken Conley Josh Faust Tully Foote Jeremy Leibs Eric Berger Rob Wheeler-Andrew Ng Morgan Quigley, Brian Gerkey. Ros: an open-source robot operating system. 2009.

[34] ROS Packages. Ros packages index. `http://www.ros.org/browse/list.php`.

[35] MoveIt. Moveit motion planning framework. `http://moveit.ros.org/`.

[36] Sebastian Starke. A hybrid genetic swarm algorithm for interactive inverse kinematics. Master's thesis, University of Hamburg, 2016.

[37] Philipp Sebastian Ruppel. Performance optimization and implementation of evolutionary inverse kinematics in ros. Master's thesis, University of Hamburg, 2017.

[38] Ompl (open motion planning library). `https://github.com/ompl/ompl`, 2025.