Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Benchmarking and Improving Inverse Kinematics for Endoscopic Inspection Tasks

vorgelegt von

Timon Engelke

Matrikelnummer 7082472

Studiengang Informatik M. Sc.

eingereicht am 11. September 2024

Betreuer: Dr. Marc Bestmann

Erstgutachter: Prof. Dr. Jianwei Zhang

Zweitgutachter: Dr. Matthias Kerzel

Abstract

In the context of robotic automation, inspection tasks are becoming increasingly important. In endoscopic inspections, a robot has to enter into a hollow cavity, such as a fuel tank. For motion planning within these constrained environments, inverse kinematics (IK) plays a crucial role. This thesis analyzes the deficiencies of current IK solvers in these environments. For this purpose, a benchmarking software and a benchmarking dataset are created. Based on two state-of-the-art solvers, BioIK and RelaxedIK, several solver improvements with a focus on endoscopic inspections are implemented and evaluated. The results show that directly including collision checking into the solvers is not feasible with the currently integrated collision detection library. The solve rates were significantly increased by implementing cost functions that instead make use of geometric properties of the problem. Additional analyses of the influence of optimization solvers and solver parameters reveal little room for improvement, while adapting the seed state shows greater potential.

Contents

1 Introduction **1**

2 Fundamentals **3**
2.1 Robotic Terms . . . 3
2.2 Forward Kinematics . . . 4
2.3 Rotation Representations . . . 5
2.4 Transformation Matrices . . . 6
2.5 Jacobian Matrices . . . 6
2.6 Inverse Kinematics . . . 7
2.7 ROS and MoveIt . . . 8
2.8 Collision Detection . . . 9
2.9 Robotic Platform . . . 11
2.10 Robotic Inspection . . . 13
2.11 Fuel Tank Maintenance . . . 14
2.12 Endoscopic Robots . . . 14

3 Related Work **17**
3.1 Inverse Kinematics Solvers . . . 17
3.1.1 KDL . . . 18
3.1.2 TracIK . . . 19
3.1.3 BioIK . . . 20
3.1.4 BioIK 2 . . . 23
3.1.5 PickIK . . . 26
3.1.6 RelaxedIK . . . 27
3.1.7 Inverse Kinematics of Minimally Invasive Surgical Robots . . . 29
3.1.8 PivotIK . . . 30
3.1.9 CuRobo . . . 31
3.2 Benchmarking . . . 32
3.2.1 IK Benchmarking . . . 32
3.2.2 Kinematic Benchmarking . . . 34

4 Benchmarking Dataset **36**

5 EBIKE **39**
5.1 Motivation . . . 39
5.2 Implementation . . . 39
5.3 Example . . . 42

6 Baseline **44**

7 Implementation **48**
7.1 MoveIt Kinematics Plugins . . . 48
7.2 RelaxedIK C++ . . . 49
7.3 BioIK . . . 51

8 IK Solver Improvements **53**
8.1 Rejection Sampling . . . 53

8.2 Collision Distance 55
8.3 Penetration Depth 57
8.4 Improving Distance Query Efficiency 59
8.5 RCM Objective 60
8.6 Collision Point Distance 62
8.7 Line Objective 64
8.8 Alignment Objective 66
8.9 Look At Objective 67
8.10 Kinematic Decoupling 68
8.11 Optimization Solvers 71
8.12 Seeding 72
8.13 Surface Inspection 74
8.14 BioIK Modes 76
8.15 BioIK Cost Functions 78
8.16 BioIK Parameters 79
8.17 Results 83

9 Conclusion **85**
9.1 Future Work 86

References **87**

Appendix A Additional Results **92**
A1 Baseline 92
A2 RelaxedIK Baseline 93
A3 Quantitative Baseline Results 94
A4 Rejection Sampling 98
A5 Penetration Depth 99
A6 Trials with High Timeout 100
A7 RCM Objective 101
A8 Collision Point Distance 101
A9 Line Objective 102
A10 Alignment Objective 102
A11 Look At Objective 103
A12 Kinematic Decoupling 103
A13 RelaxedIK Cost Functions 104
A14 Surface Inspection 104
A15 Final Results on HydrogenTankSmall 105

Appendix B Open Source Contributions **107**

List of Figures

Figure 2.1: Two-dimensional robotic arm illustrating basic concepts 3
Figure 2.2: Forward kinematics illustrated on a two-dimensional robotic arm 4
Figure 2.3: Inverse kinematics illustrated on a two-dimensional robotic arm 7
Figure 2.4: ROS logo 8
Figure 2.5: Depiction of a joint in a URDF 9
Figure 2.6: Rendering of ELISE on a UR10 arm 11
Figure 2.7: Closeup of ELISE's joints 11
Figure 2.8: Rendering of ELISE entering a fuel tank 12
Figure 2.9: Eeloscope2 on UR10e arm inspecting an aircraft wing 12
Figure 2.10: Aircraft fuel tank inspection 14
Figure 2.11: Da Vinci Xi, a robot used in robot-assisted minimally invasive surgery .. 14
Figure 2.12: Remote center of motion in a surgical robot 15
Figure 3.1: Family tree of IK solvers 17
Figure 3.2: BioIK evolutionary algorithm 21
Figure 3.3: Loss functions used in the RelaxedIK family 28
Figure 3.4: Example benchmark of inverse kinematics with an RCM 33
Figure 3.5: Kinematic benchmarking of a robot operating on a plane 34
Figure 4.1: Scenarios in the evaluation dataset 36
Figure 5.1: EBIKE logo 39
Figure 5.2: UML class diagram of EBIKE 40
Figure 5.3: Entity-Relationship Model of EBIKE 41
Figure 5.4: EBIKE output for UR10 on two different scenarios 42
Figure 6.1: Cumulative solve rates for the *Random* scenario 44
Figure 6.2: Partial coverage (80.4%) of *Table* using TracIK 46
Figure 6.3: Cumulative solve rates for the *Table* scenario 47
Figure 6.4: Solver performance on *Barrel* and *Shelf* scenarios 47
Figure 6.5: Hydrogen tank scenarios 47
Figure 7.1: Class diagram of RelaxedIK 50
Figure 7.2: Histogram of objective function calls for BioIK and RelaxedIK 50
Figure 7.3: Comparison of RelaxedIK to other MoveIt IK solvers 51
Figure 8.1: Results of IK solvers with and without rejection sampling 54
Figure 8.2: Penetration depth results on *HydrogenTank* 57
Figure 8.3: Penetration depth and collision distance solvers with 10s timeout 58
Figure 8.4: Illustration of different targets solved by traversing an RCM 60
Figure 8.5: Results of RCMGoal on *HydrogenTank* and *HydrogenTankSmall* 60
Figure 8.6: Comparison of different RCM goals on *HydrogenTank* 61
Figure 8.7: Comparison of RCM goals on *HydrogenTank* and *HydrogenTankSmall* 62
Figure 8.8: Illustration of the line objective, keeping a link on the red line 64
Figure 8.9: Results of the line objective on both fuel tanks 65

Figure 8.10: Targets reached by RelaxedIK using the LineGoal 65
Figure 8.11: Example solution for an unreachable target using the LineGoal 65
Figure 8.12: Illustration of the alignment objective 66
Figure 8.13: Results of the alignment objective on *HydrogenTank* 66
Figure 8.14: Illustration of the look at objective 67
Figure 8.15: Results of the look at objective with BioIK 67
Figure 8.16: Illustration of kinematic decoupling 68
Figure 8.17: Illustration of the distal solver goal 70
Figure 8.18: Results of kinematic decoupling 70
Figure 8.19: Different optimization solvers on three problems 71
Figure 8.20: Seed position used for *HydrogenTank* 72
Figure 8.21: Solver performance when a seed is used on *HydrogenTank* 72
Figure 8.22: Targets reached by KDL with a seed within the first solver iteration 73
Figure 8.23: Seed variants for BioIK on *HydrogenTank* 73
Figure 8.24: Illustration of inspection criteria 75
Figure 8.25: RelaxedIK with ScanGoal on *Random* and *HydrogenTank* 75
Figure 8.26: Comparison of different modes of BioIK 2 76
Figure 8.27: Comparison of the best geometry-based goals for *HydrogenTank* 84
Figure A1: Cumulative solve rates for other scenarios 92
Figure A2: Comparison of RelaxedIK to existing solvers on additional scenarios 93
Figure A3: Results of penetration depth goal on *TableObjects* 99
Figure A4: Results of trials with 10s timeout 100
Figure A5: Comparison of different RCMGoals on *HydrogenTankSmall* 101
Figure A6: ScanGoal on *HydrogenTankSmall* 104
Figure A7: Comparison of the best geometry-based goals for *HydrogenTankSmall* ... 106

List of Algorithms

Algorithm 3.1: KDL algorithm 18
Algorithm 3.2: Termination metric used in TracIK 20
Algorithm 3.3: BioIK algorithm 21
Algorithm 3.4: BioIK 2 algorithm 24

List of Tables

Table 3.1: BioIK modes 25
Table 3.2: Evaluation metrics of inverse kinematics papers 33
Table 6.1: Solve rates after given durations and solver iterations on *Random* 45
Table 7.1: Configurable parameters for BioIK 52
Table 8.1: Results of IK solvers with and without rejection sampling on *Random* 54
Table 8.2: Mean solver iterations with and without rejection sampling on *Random* ... 55
Table 8.3: Time spent on collision checking in RelaxedIK 56
Table 8.4: Time spent on penetration depth checking in RelaxedIK 57
Table 8.5: Time spent on collision point distance in RelaxedIK 63
Table 8.6: Quantitative results of LineGoal on *HydrogenTank* 64
Table 8.7: Quantitative results for AlignmentGoal on *HydrogenTank* 66
Table 8.8: Success rates for the first iteration with a seed on *HydrogenTank* 73
Table 8.9: Results of BioIK with and without an IKCostFn on *Random* 79
Table 8.10: BioIK parameter ranges used for parameter optimization 79
Table 8.11: Results of parameter optimization for BioIK on *Random* 80
Table 8.12: Results of parameter optimization for BioIK on *TableObjects* 81
Table 8.13: Results of parameter optimization for BioIK on *HydrogenTank* 81
Table 8.14: Overview of the results on *HydrogenTank* 83
Table A1: Baseline on *Barrel* 94
Table A2: Baseline on *Shelf* 94
Table A3: Baseline on *Table* 95
Table A4: Baseline on *SmallTable* 95
Table A5: Baseline on *TableObjects* 96
Table A6: Baseline on *HydrogenTank* 96
Table A7: Baseline on *HydrogenTankSmall* 97
Table A8: Base results on *HydrogenTank*, all solvers use rejection sampling 98
Table A9: Base results on *HydrogenTankSmall*, all solvers use rejection sampling 98
Table A10: Results of collision point distance (RelaxedIK) 101
Table A11: Results of the line objective on *HydrogenTank* 102
Table A12: Results of the line objective on *HydrogenTankSmall* 102
Table A13: Results on the alignment objective on *HydrogenTank* 102
Table A14: Results on the alignment objective on *HydrogenTankSmall* 103
Table A15: Quantitative results for look at objective (BioIK) 103
Table A16: Results of kinematic decoupling 103
Table A17: Results of RelaxedIK with and without an IKCostFn on *Random* 104
Table A18: Overview of the results on *HydrogenTankSmall* 105

Chapter 1
Introduction

In recent years, robots have become widely used across industries to automate processes and reliably perform repetitive tasks. Often, tasks that are difficult to perform for a human or that pose a risk to the worker's safety are of special interest for robotic automation. These tasks can then be performed by an automated, semi-automated, or remote-controlled robot. In each case, a crucial part of the task is motion planning, meaning that a plan for the movement of the robot's joints has to be created. Most robots consist of a series of links and joints, called the kinematic chain. In motion planning, the movement of the robot's end effector, i. e., the last link in the kinematic chain, plays an important role because a tool or sensor is often attached to it. Finding the joint configuration to attain a certain end effector pose is called inverse kinematics (IK).

The opposite problem of inverse kinematics is forward kinematics, the task of calculating an end effector pose given the robot's configuration. While forward kinematics is easy to solve by applying the offsets and rotations for each motor to the pose of the base link, the inverse kinematics problem is much harder. In general, no closed-form solution exists and certain target poses can be unsolvable, for example when not enough degrees of freedom are available to achieve a given pose, or non-unique, when multiple configurations exist that result in the same end effector pose.

While many different inverse kinematics solvers exist, they are often evaluated in simple scenarios that do not have a high number of obstacles in the environment. These algorithms perform badly in highly constrained environments because they do not account for collisions during the optimization routine. Instead, the validity of a solution is only checked after a solution has been found.

Collision-aware inverse kinematics solvers play a role in many types of problems, such as gripping an object from a full shelf, navigating an arm around an occluded table, or performing inspection in narrow spaces. This thesis focuses on inverse kinematics in endoscopic inspection environments where robots are deployed to enter a hollow space through a small opening for inspection or repair. The primary subject of this work is ELISE, an endoscopic robot that is currently under development at the Institute for Maintenance, Repair, and Overhaul of the German Aerospace Center (DLR). It is developed to inspect parts of planes, for example fuel tanks or jet engines, especially in the context of hydrogen-powered aviation, and consists of three metal rods connected by revolute joints with a camera or tool mounted at the tip. By nature, endoscopic inspection tasks take place in highly constrained environments: Only a very limited set of robot configurations can pass through the opening without colliding with the environment, for example the fuel tank. In the process of automating these tasks, it is pertinent to autonomously find robot configurations that reach certain poses inside the tank.

This thesis aims to answer the following two research questions:

1. What automated methods can be used to evaluate inverse kinematics solvers?
2. Which approaches are effective in improving inverse kinematics solvers for endoscopic inspections?

To evaluate the performance of existing solvers on these tasks, a benchmarking dataset with non-endoscopic and endoscopic scenarios is created. Then, several improvements to current inverse kinematics solvers are implemented and evaluated on the created benchmarking dataset.

The implementation is done for the ROS 2 framework [Mac+22], a framework that is widely used in robotics, especially in the research community. It uses the MoveIt [Chi16] library which is a motion planning library in ROS 2.

The thesis is structured in the following way. The next chapter introduces the fundamentals required for understanding the subsequent chapters. In Chapter 3, existing inverse kinematics solvers are presented and their advantages and disadvantages are highlighted. Existing inverse kinematics and kinematic structure benchmarks are described. The benchmarking dataset is introduced in Chapter 4, along with a description of the process of its creation. As part of this thesis, a benchmarking library called EBIKE is developed and described in Chapter 5. Chapter 6 presents initial evaluations of current IK solvers on the dataset, while Chapter 7 details the implementation process. In Chapter 8, the various possible improvements to inverse kinematics solvers are described and evaluation results are presented and discussed. The thesis concludes with a summary of the findings in Chapter 9.

Chapter 2

Fundamentals

This master's thesis focuses on the kinematics and control mechanisms of robotic systems. Accordingly, it is essential to define important terms used throughout the thesis. The chapter starts by explaining key robotic concepts, including basic terminology, forward and inverse kinematics, and rotation representations. Then, the libraries that are used in the implementation are presented. The robotic platform, ELISE, is introduced, and an overview of applications for endoscopic robots is given.

2.1 Robotic Terms

A robot is "a machine that can perform a complicated series of tasks by itself."[1] It consists of a series of **joints**, typically articulated by servo motors, which are connected by **links** [SK16]. The links are assumed to be **rigid**. The most important types of joints are **revolute joints** and **linear joints**. While revolute joints allow movement around an axis, linear joints permit movement along an axis. Joints have **joint limits** that determine how far the joint can move. The positions of all joints determine the **configuration** of a robot. The minimum number of variables required to describe the configuration of a robot is called its **degrees of freedom**. If the joints are independent, the degree of freedom is equivalent to the number of joints. An illustration of a simple robotic arm with three links and joints is shown in Figure 2.1.

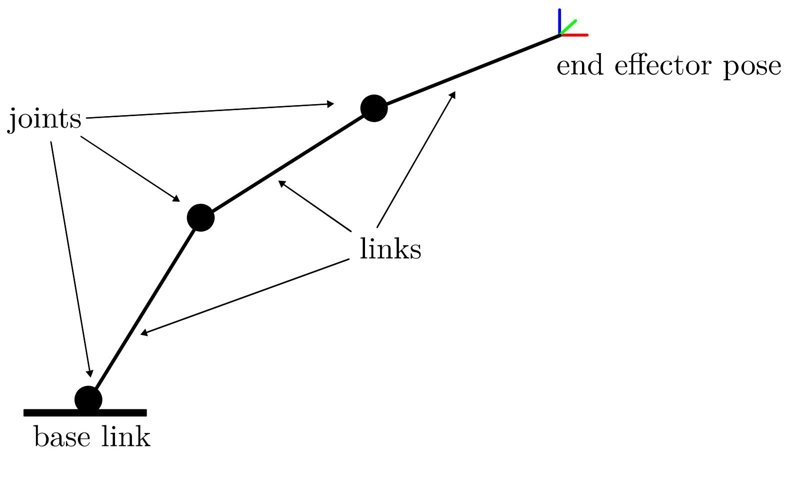


Figure 2.1: Two-dimensional robotic arm illustrating basic concepts

[1]https://www.oxfordlearnersdictionaries.com/definition/english/robot, accessed 24/04/2024

The series of links and joints of a robot is called a **kinematic chain**, with the final link being called the **end effector**. The end effector is particularly relevant as it interacts directly with its environment using mounted tools or sensors. A robot can have multiple kinematic chains and therefore multiple end effectors. In three-dimensional space, also called **Cartesian space**, an object's **pose** is defined by its **position** and **orientation**. Position is typically expressed as a set of x, y, and z coordinates, while orientation can be represented using various methods, including rotation matrices, quaternions, axis-angle descriptions, or Euler angles (see Section 2.3). The distance between two positions is the **linear distance**, the norm of the vector connecting them. The distance between two orientations, the **angular distance**, is the angle of rotation required to rotate one rotation to the other. The pose of a robot's end effector can be expressed either in Cartesian space, stating its position and orientation, or in joint space, specifying the angles of the joints in the kinematic chain. The choice of representation depends on the nature of the conducted task. In this thesis, the focus is on converting one into the other.

2.2 Forward Kinematics

In robotics, forward kinematics describes the problem of calculating the pose (position and orientation) of a robot's end effector from the configuration of its joints and its kinematic structure. The calculation is straightforward. Each joint is defined by its offset to its parent joint and its applied transform. The offset is a fixed transform between the two joints that depends on the kinematic structure of the robot. The applied transform of a joint depends on its configurations. Starting with the pose of its base link, the offsets and transforms of all joints can be applied sequentially [SK16]. The equation for forward kinematics is generally expressed as $x = f(q)$ where q is the vector representing the robot's configuration (the values of the joints) and x is the vector giving the end effector pose. Figure 2.2 illustrates the problem on the two-dimensional robot arm where the pose of the end effector depends on the three joint positions q_0, q_1, and q_2.

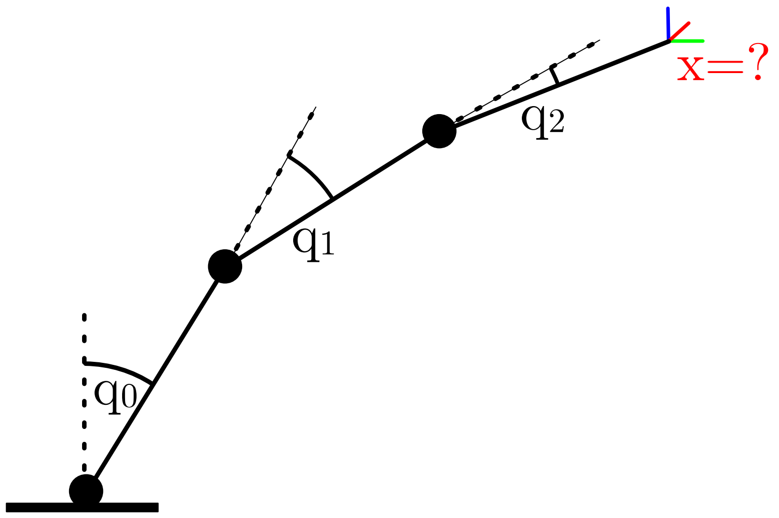


Figure 2.2: Forward kinematics illustrated on a two-dimensional robotic arm

2.3 Rotation Representations

Rotations can be represented in several different ways. The most intuitive way of defining them is using Euler angles. Euler angles are a vector of three angles that describe three successive rotations. The axes around which the rotations are performed must be given with the representation. Take, for example, the order Z-Y-X, then, the first angle describes the rotation around the Z axis, the second angle describes the rotation around the *rotated* Y axis, and the last angle then rotates around the twice rotated X axis. The rotations can also be performed around non-moving axes, this representation is called fixed angles. In addition to being ambiguous because of the required axis order, both Euler angles and fixed angles suffer from the problem that the first two rotations can occur in such a way that the third rotation is around an axis that was already a previous rotation axis; this is called a gimbal lock or representation singularity. [SK16]

Another rotation representation is rotation matrices. The rotation described by such a matrix is applied by multiplying the matrix with a vector to be rotated. The rotation matrix is a three-by-three matrix whose columns are orthonormal vectors that give the new axes after the rotation in terms of the old axes. Convenient properties of rotation matrices are that rotations can be concatenated by multiplying their rotation matrices and that the inverse of a rotation matrix is its transpose. [SK16]

By Euler's rotation theorem [Eul76], any rotation or sequence of rotations can be described as a single rotation around a fixed axis. This description makes it possible to describe a rotation by this axis and the angle of rotation, called the axis-angle notation, represented by a unit vector $\hat{w} = \left(w_x, w_y, w_z\right)^T$ pointing along the axis of rotation and the angle of rotation θ [SK16]. When the axis-angle notation is used to describe the difference between two rotations, θ is their angular distance. The axis-angle values can also be combined by multiplying the axis and the angle, leading to a three-variable representation where the vector norm gives the angle of rotation. Whenever the function AXIS-ANGLE is used in the remainder of this thesis, it refers to this three-variable representation.

Finally, a quaternion ε [Ham44] is composed as $\varepsilon = \varepsilon_0 + \varepsilon_1 i + \varepsilon_2 j + \varepsilon_3 k$ where i, j, k are operators such that

$$\begin{aligned} ii &= 1, & ij &= k, & ji &= -k, \\ jj &= 1, & jk &= i, & kj &= -i, \\ kk &= 1, & ki &= j, & ik &= -j \end{aligned}$$

The conjugate of a quaternion is defined as $\tilde{\varepsilon} = \varepsilon_0 - \varepsilon_1 i - \varepsilon_2 j - \varepsilon_3 k$. Unit quaternions are quaternions for which the product with their conjugate is equal to 1, i. e., the sum of their squared parameters is equal to 1. They can be used to describe rotations and are obtained from an axis-angle pair in the following way:

$$\varepsilon_0 = \cos\frac{\theta}{2} \qquad \varepsilon_1 = w_x \sin\frac{\theta}{2} \qquad \varepsilon_2 = w_y \sin\frac{\theta}{2} \qquad \varepsilon_3 = w_z \sin\frac{\theta}{2}$$

Conveniently, concatenation of two rotations can be done by multiplying their quaternions, and the inverse of a unit quaternion is, by definition, its conjugate. [SK16]

2.4 Transformation Matrices

Any rigid transformation, i. e., a translation and rotation in space without deformation, can be represented as a transformation matrix.

$$T = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

R is a 3×3 rotation matrix, t is a 3×1 translation vector. The inverse of a transformation matrix can easily be calculated.

$$T^{-1} = \begin{pmatrix} R^T & -R^T t \\ 0 & 1 \end{pmatrix}$$

Consecutive transformations are defined by the product of their transformation matrices.

The set of all transformation matrices forms a Lie group [LE93] which is called SE(3). An element of SE(3) can be transformed to the Lie algebra $\mathfrak{se}(3)$. The Lie algebra $\mathfrak{se}(3)$ is represented by a six-dimensional twist vector that describes velocities along and around an axis required to reach the transformation represented by an element of SE(3). The function that maps elements from SE(3) to $\mathfrak{se}(3)$ is called the logarithm map. [Sel05]

2.5 Jacobian Matrices

In robotics, the Jacobian is the matrix of all partial derivatives of the forward kinematics function.

$$J(q) = \begin{pmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \cdots & \frac{\partial x}{\partial q_n} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \cdots & \frac{\partial y}{\partial q_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \varphi}{\partial q_1} & \frac{\partial \varphi}{\partial q_2} & \cdots & \frac{\partial \varphi}{\partial q_n} \end{pmatrix}$$

It is a $m \times n$ matrix where m is the number of degrees of freedom of the end effector and n is the number of joints. In the context of this thesis, m is always six because the end effector has six variables, three for its position in three-dimensional space and three for the orientation.

The Jacobian matrix can be used to calculate the forward dynamics, i. e.,

$$\dot{x} = J(q)\dot{q}$$

Here, $\dot{x}$ is the velocity of the end effector and $\dot{q}$ is the velocity of the joints.

The Jacobian describes how the movement of the joints influence the movement of the end effector. Therefore, it can also be used to analyze the range of motion that can be covered in a given configuration. If some velocities in Cartesian space are not achievable with any joint velocities, the configuration is called singular and its Jacobian matrix does not have full rank. As an example, if it is impossible to move any joint such that the end effector moves in positive x direction, the configuration is singular.

The joint velocity vectors can be divided into two linear subspaces: the range space and the null space. The null space is defined as $N = \{\dot{q} \mid J(q)\dot{q} = 0\}$, the joint velocities that have no influence on the Cartesian space velocities of the end effector. All other velocity vectors influence the end effector velocities and are called the range space. A three-dimensional robot that has more than six degrees of freedom always has a non-empty null space because there are joint movements that can be negated by the movement of other joints. A singularity further increases the size of the null space and reduces the range space because more joint movements cannot influence the end effector movement. Since the velocities in the null space do not influence the end effector velocity, they can also be used to achieve other goals such as obstacle or singularity avoidance.

For singularity avoidance, three measures are used to define the distance to a singularity. The first is the manipulability measure $\mu = \sqrt{|JJ^T|}$ which is zero at singularities. The second measure is the condition number $\kappa = \frac{\sigma_1}{\sigma_m}$, i. e., the quotient of the smallest and largest singular value of the Jacobian. It is infinite at singularities because the smallest singular value becomes zero. As a third measure, the smallest singular value σ_m can be used directly. [SK16]

Obstacle avoidance could for example be achieved by searching the null space for collision-free solutions, but the potentially large size of the null space makes this approach unpractical.

2.6 Inverse Kinematics

Inverse kinematics (IK) is the opposite of forward kinematics. Mathematically, it can be expressed as $q = f^{-1}(x)$, where q denotes the joint configuration needed to achieve a specific end effector pose x. Figure 2.3 illustrates the IK problem on the two-dimensional robot arm. The pose x of the end effector is given and the joint values q_0, q_1, and q_2 required to reach this position have to be determined.

In practice, this is useful for a variety of tasks. For example, a robot could determine the pose of an object in space via its camera, then calculate a pose where its hand could grasp the object and finally move the hand to that pose. For the last step, calculating how its joints have to be configured to reach that pose, inverse kinematics are required. Extending the example, the robot has to move from its current pose to the target pose.

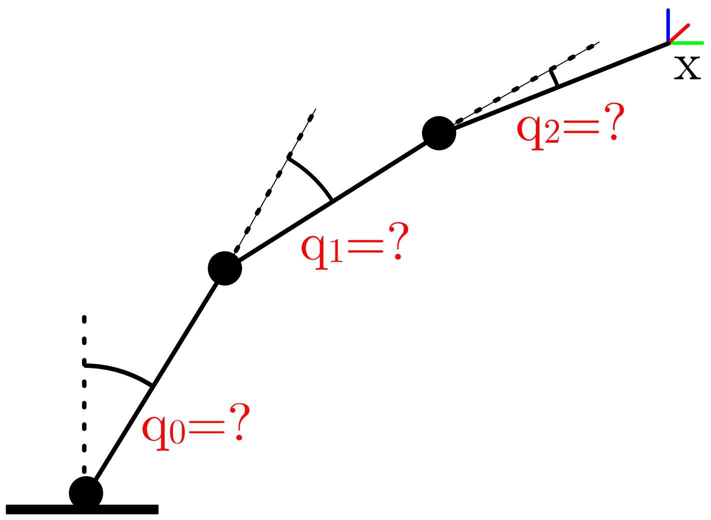


Figure 2.3: Inverse kinematics illustrated on a two-dimensional robotic arm

For this movement, a **trajectory**, i. e., a path that the hand follows over time, has to be planned and executed. A simple approach is to create a trajectory in joint space. This means that the interpolation forming the trajectory happens for each joint independently. The resulting movement is often unnatural as the end effector moves on a complicated trajectory in Cartesian space. An alternative is to compute a trajectory in Cartesian space. This requires interpolation in Cartesian space and solving inverse kinematics for each step of the path. While this can result in a more natural movement, other issues may arise, e. g., because joint limits are reached or jumps in joint space that are not executable by the robot happen.

A problem with inverse kinematics is that while forward kinematics are simple to calculate and a single solution exists, for the inverse problem, multiple solutions may exist, i. e., multiple configurations exist where the same end effector pose is reached, or no solutions might exist, for example, because the end effector pose is not reachable.

The two main types of inverse kinematics solvers are analytical and numerical solvers [SK16]. **Analytical solvers** use a closed-form solution adapted to a given robot that directly returns the required joint positions given an end effector pose. They can either be formed based on the geometric properties of the robot or algebraic considerations of the robot's kinematics. Geometric solutions are often a result of trigonometric equations that can be found in the robot's kinematic structure or other geometric properties, such as intercepting or parallel lines. Algebraic solvers are created by mathematically inverting the forward kinematics equations. Unfortunately, both the geometry of a robot and the forward kinematics equations become more complex with increasing complexity of the robot, for example more degrees of freedom or a more difficult kinematic structure. Therefore, for a lot of robots, no analytical solution is known and it might not even exist. However, if the exist, analytical inverse kinematics solvers find solutions extremely fast (similar to forward kinematics) because they only have to solve a set of equations.

The other group of solvers are numerical solvers. **Numerical solvers** do not directly find an optimal solution. Instead, they iteratively improve solutions until a termination criterion is reached. The two principal approaches to numerical solvers are Inverse Jacobian-based methods and optimization-based methods. Other approaches include evolutionary algorithms which represent candidate solutions as individuals that reproduce over generations and memetic algorithms which combine evolutionary and optimization algorithms. For some of these inverse kinematics solvers, it is also possible to specify additional cost functions that define costs for positional or rotational errors or other criteria such as collisions, joint displacements, etc. Examples of these cost functions are given in Sections 3.1.3 and 3.1.6.

2.7 ROS and MoveIt

The Robot Operating System (ROS) is a middleware used in robotics. It is widely used in research and industry for a number of different robots, such as quadruped robots, submarine robots, drones [Mac+22], and even soccer-playing robots [BHW17]. The software platform provides the means to split the software running on a robot in cohesive, modular nodes that communi-

Figure 2.4: ROS logo[2]

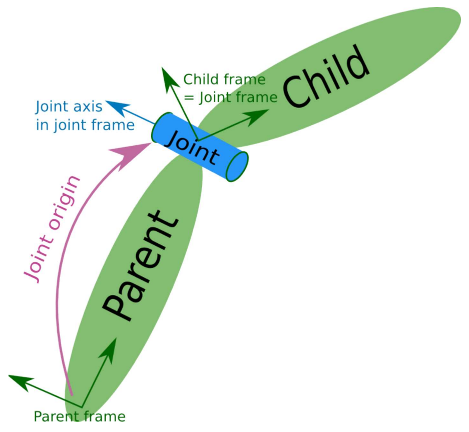


Figure 2.5: Depiction of a joint in a URDF[5]

cate via message exchange. The current version of the project, ROS 2, focuses on security, reliability, and large scale systems by basing the communication on the Data Distribution Service (DDS). ROS is developed open source and supported by a large community from research and industry.

MoveIt is software for “motion planning, manipulation, 3D perception, kinematics, control, and navigation” [Chi16]. It is widely used and can handle many different robots. For kinematics, a plugin-based architecture is used that supports different solver implementations for different robots. By default, a numerical solver for inverse kinematics is used. MoveIt is currently stewarded by PickNik Robotics and supported by a “community of collaborators and maintainers”[3].

An alternative to MoveIt is Tesseract [Arm24], a relatively new motion planning framework with integration into ROS 2 developed by the Southwest Research Institute. At the point of writing, the package is still “under heavy development and subject to change”[4] and the documentation is practically nonexistent, disqualifying the library for use in this thesis.

In ROS 2, robots are represented using the unified robot description format (URDF). In a URDF file, a kinematic chain is defined as a series of joints that each have a transform T_{origin} from the previous joint and apply a transform T_{joint} that depends on the joint configuration. For revolute joints, for example, T_{joint} is a rotation around a single axis. Figure 2.5 shows a sketch of such a joint, with its origin relative to the parent joint and its rotation around an axis. The URDF describes a transform from a base to an end effector that is composed of several joint transformations.

2.8 Collision Detection

Collision detection is the problem of detecting whether two or more objects in space collide, i. e., they partially occupy the same space. Related, **proximity computation** describes the process of computing the shortest distance between any two points of two objects. It is sometimes also called separation distance query or **collision distance query**.

[2]License: CC-BY-NC 4.0, https://github.com/ros-infrastructure/artwork
[3]https://picknik.ai/moveit/, accessed 13/08/2024
[4]https://github.com/tesseract-robotics/tesseract/blob/0.24.1/README.md
[5]http://wiki.ros.org/urdf/XML/joint, accessed 15/08/2024

Penetration depth refers to the minimum translation movement required to move two colliding objects into non-collision.

In general, collision detection is split into a broad phase and a narrow phase. In the broad phase, objects that may collide are pre-selected and objects that do not collide are excluded. This can for example be done by wrapping the objects in bounding boxes and checking for collisions between them. This broad phase detection is usually very fast because it does not require a precise analysis of the geometry of the objects. If and only if two objects are detected as colliding in the broad phase, the pair is passed on to the narrow phase. In this phase, exact collision detection is performed, taking into account the precise meshes of the objects. A mesh is a collection of vertices, edges, and faces that define the shape of an object. [Eri05]

The Flexible Collision Library (FCL) [PCM12] is a general-purpose collision detection and proximity query library and is used by default in MoveIt. It supports different types of objects and different algorithms based on the object types. For broad-phase collision detection, FCL uses Sweep and Prune. This algorithm uses axis-aligned bounding boxes and detects collisions by checking for overlap between the bounding box intervals on each axis separately. Objects are detected as colliding when their intervals overlap on all three axes. The algorithm has an expected runtime in $O(n)$ for n objects by keeping the interval lists sorted.

In the narrow phase, FCL uses the Gilbert-Johnson-Keerthi (GJK) algorithm [GJK88] for collision detection and distance calculation and the Expanding Polytope Algorithm (EPA) [Van01] for penetration depth computation. The GJK algorithm works by iteratively improving a geometric representation of the overlap of two objects until it determines whether they intersect. For non-convex or non-polytope objects, FCL uses Bounding Volume Hierarchies (BVHs). Alternatives are spatial decomposition techniques, for example kd-trees or octrees, but the authors claim that BVHs are faster [PCM12].

The library also supports continuous collision checking, i. e., checking whether moving objects collide at any point in time. Since exact penetration depth computation has a very high complexity ($O(n^6)$ for non-convex objects [Kim+03]), FCL only approximates the penetration depth for mesh-based models by calculating the penetration depth between the colliding triangles of the meshes, giving only a lower bound of the actual penetration depth.

While GPU support and parallelization can significantly improve collision detection speed, neither is supported by FCL. The authors of FCL developed an algorithm in 2012 that uses GPU-based parallel collision detection, greatly improving the number of collision queries that can be executed per second [PM12]. However, the code is not integrated into any library.

HPP-FCL [Pan+15] is a newer collision library by FCL's authors that improved the performance of the GJK algorithm by a factor of two [Mon+22]. As in the original GJK algorithm, it only supports convex objects.

Instead of FCL, MoveIt also supports using Bullet for collision detection. Many Bullet features, such as parallelization and GPU usage, are not supported by MoveIt[6]. To date, there are no ongoing plans to integrate GPU usage into MoveIt[7].

2.9 Robotic Platform

The principal robot in this thesis is ELISE which is developed by the Institute for Maintenance, Repair, and Overhaul of the German Aerospace Center (DLR). Since its development is not completed at the point of writing this thesis, the kinematic model of the entire robot is only available in simulation. Figure 2.6 provides a rendering of the robot, illustrating its current design.

ELISE has three revolute joints that connect parts of a thin metal rod. They are actuated using an antagonistic rope mechanism. The controlling motors are located in a box at the

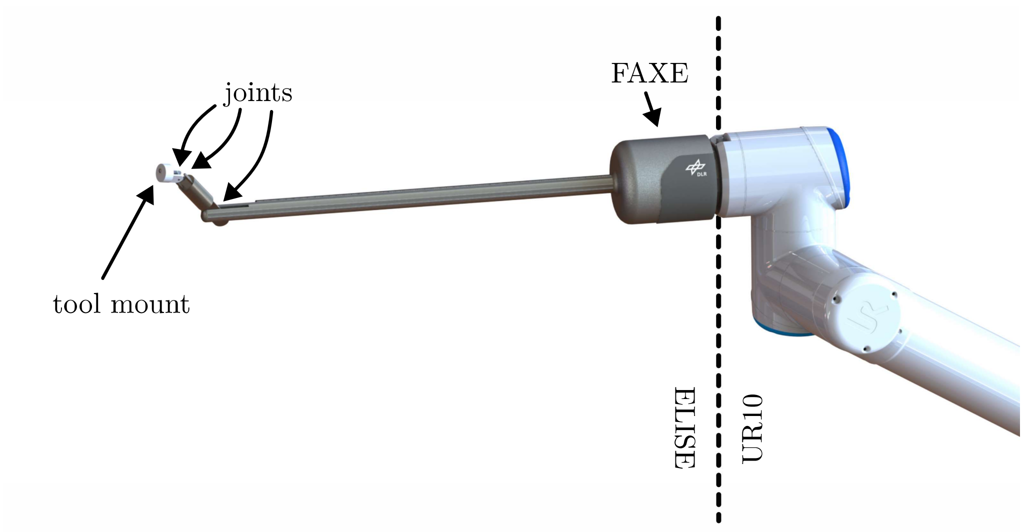


Figure 2.6: Rendering of ELISE on a UR10 arm (created by Florian Heilemann, DLR), annotated

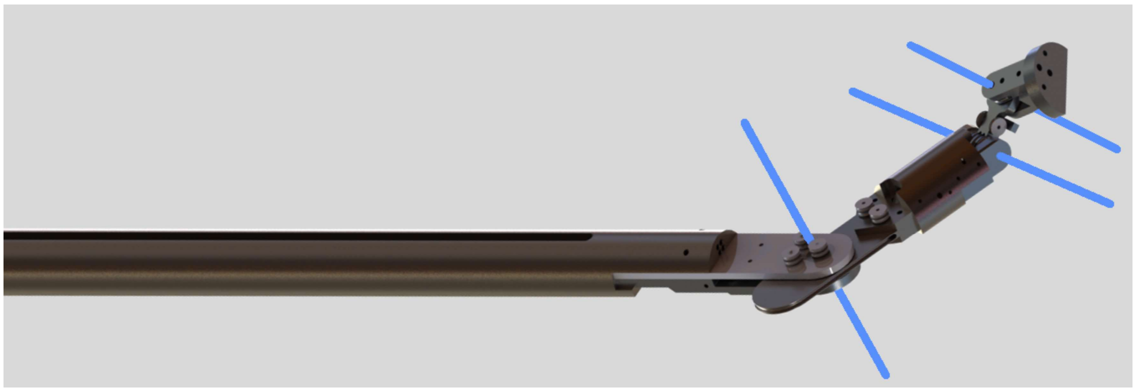


Figure 2.7: Closeup of ELISE's joints (rendering created by Florian Heilemann, DLR), joint rotation axes are added in blue

[6]https://github.com/moveit/moveit/issues/1646, accessed 09/08/2024
[7]https://discourse.ros.org/t/gpu-acceleration/31611/3, accessed 09/08/2024

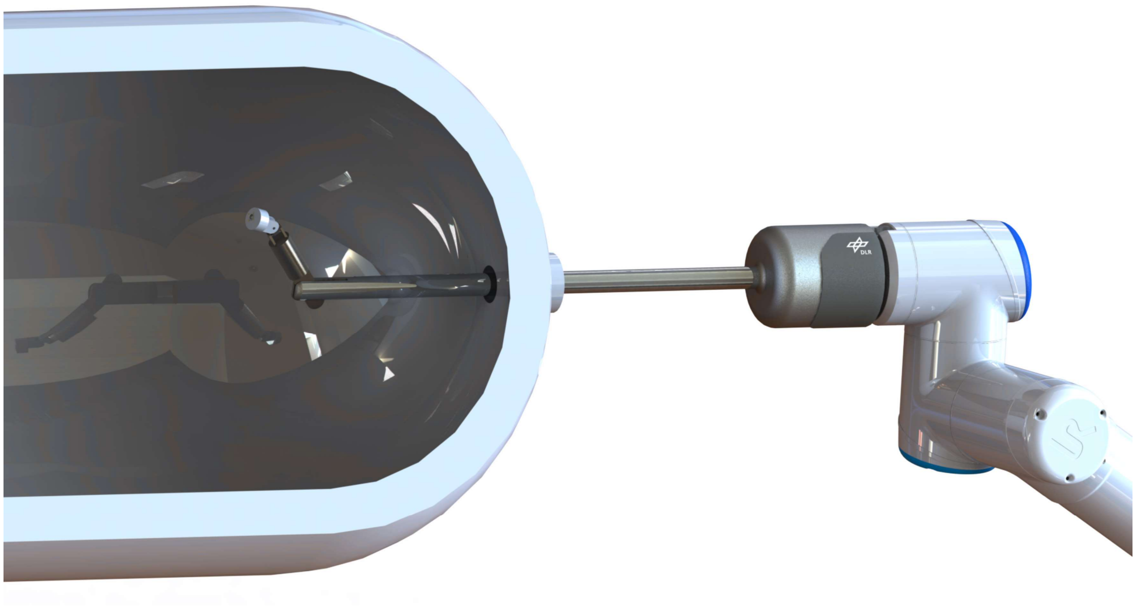
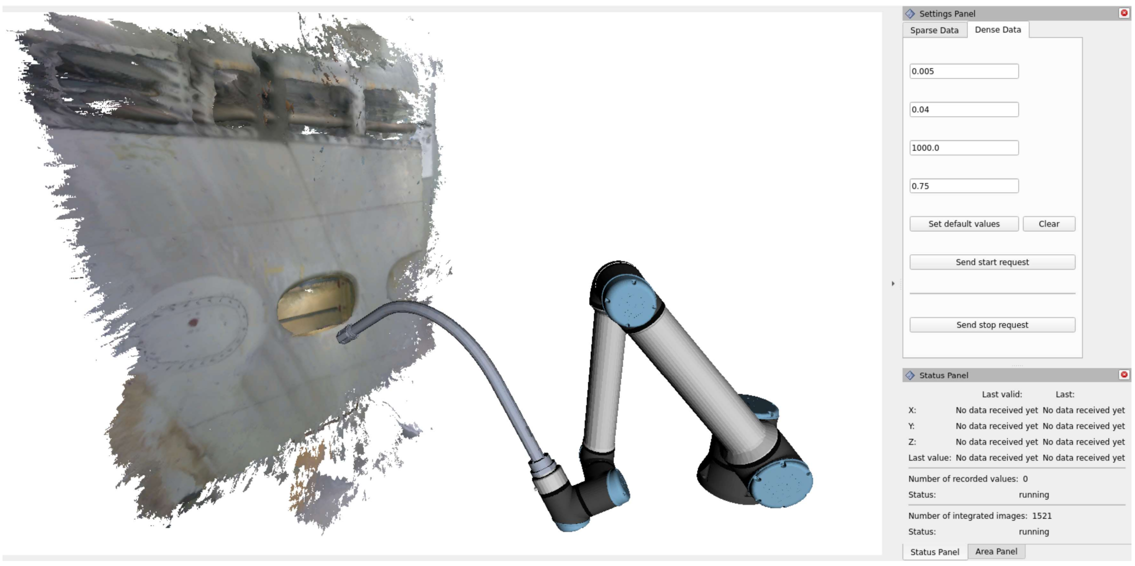


Figure 2.8: Rendering of ELISE entering a fuel tank (created by Florian Heilemann, DLR)

base of ELISE, called FAXE. For each joint, two ropes are directed through the shaft of ELISE and connected to the two motors controlling the joint. To move the joint, one of the motors pulls on one rope while the other one slowly releases the other in a controlled way, holding it such that the joint does not have any slack. Figure 2.7 shows a closeup of ELISE's joints and the axes of rotation.

On the tool mount at the end of ELISE, different sensors or manipulators, such as a camera, an ultrasonic sensor, a welding unit, or a gripper, can be attached. ELISE's shaft is horizontally divided into two parts, the bottom part is used for the rope mechanism, and the top part is left for wiring of the attached tool. ELISE itself can be attached to



```
0.005
0.04
1000.0
0.75
```

```
            Last valid:           Last:
X:          No data received yet  No data received yet
Y:          No data received yet  No data received yet
Z:          No data received yet  No data received yet
Last value: No data received yet  No data received yet
Number of recorded values:  0
Status:                     running
Number of integrated images:  1521
Status:                     running
```

Figure 2.9: Eeloscope2 on UR10e arm inspecting an aircraft wing [Ric24]

different bases as an end-of-arm tool. One possible base for ELISE is the UR10 arm, a robotic arm with six degrees of freedom.

The practical use of ELISE is to perform endoscopic tasks because the kinematic structure makes it possible to enter through small maintenance ports and place instruments in constrained spaces. These tasks are concentrated on the maintenance, repair, and overhaul (MRO) sector and can include scanning the object to find fissures or repairing the object by welding. The objects on which ELISE's design is focused are related to transport, mainly hydrogen fuel tanks for use in storage or automotive vehicles and jet turbines. Figure 2.8 shows the robot entering a hydrogen tank through a maintenance port.

Another robot was developed during the FuTaMa2 (Fuel Tank Maintenance 2) project at the same institute, the Eeloscope2 robot. It can also be attached to a UR10 robotic arm and was designed for inspection tasks, with three RGB-D cameras mounted at its end effector. It consists of a curved but fixed shaft, enabling it to enter through some openings like aircraft maintenance ports, but also restricting its range compared to ELISE. Figure 2.9 shows a rendering of the robot.

2.10 Robotic Inspection

Robotic inspections are used in many different fields, such as civil infrastructure, ships, turbines, and aircraft [Alm+16]. These inspection tasks are often dangerous for humans and successful spotting of damages is critical for the system's safety. Automation using robots can increase the safety of these inspections by introducing repeatable and quantifiable methods.

Generally, an inspection consists of three parts: coverage path planning, model reconstruction, and inspection. In coverage path planning, viewpoints are selected from which all parts of the object are visible and a trajectory is planned between them. In the next step, a model of the object is generated from the data accumulated in the first step. Finally, the actual inspection can take place using, for example, anomaly detection algorithms [RM23].

Depending on the inspected object, a wide range of sensors can be used. Next to videos or still cameras, crucial for visual inspection, ground-penetrating radar, infrared, sonar, lidar, gyroscopes, and microphones are other sensors used in non-destructive testing. More specific sensors, such as seismic sensors, methane gas sensors, or long wavelength IR for humidity detection are also used in some cases. [NJ95]

An advantage of automated inspection is that quantitative and qualitative information can be collected and stored side-by-side, making inspection results more repeatable and reliable. Storing inspection results also makes it possible to track the maintenance and defect history of an object and facilitate repairs [HDW21]. In aircraft MRO, this information can, for example, be integrated into data aggregation tools such as Airbus Skywise[8] or Lufthansa AVIATAR[9] ("Digital Twins").

[8]https://aircraft.airbus.com/en/services/enhance/skywise, accessed 20/08/2024

[9]https://www.lufthansa-technik.com/en/aviatar, accessed 20/08/2024

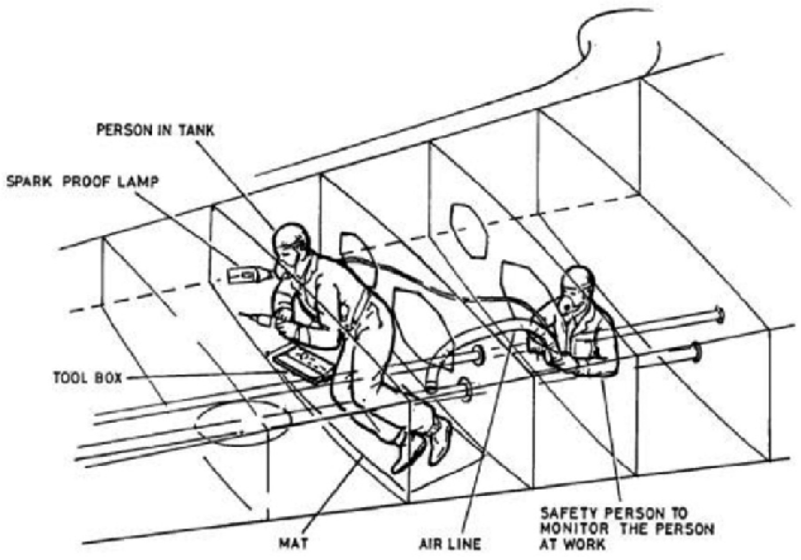


Figure 2.10: Aircraft fuel tank inspection [NWX18]

2.11 Fuel Tank Maintenance

Regular maintenance checks are scheduled to ensure the safety of an aircraft. A crucial part of these checks is fuel tank maintenance, as fuel tanks are highly safety-critical because of the flammable fuel contained within them. Important aspects of a fuel tank system are leakage protection, explosion protection, structural strength, and cleanliness [HDW21]. For this reason, the European Union Aviation Safety Agency and the Federal Aviation Administration have published regulations dictating strict fuel tank maintenance checks [Fed01]. During these checks, General Visual Inspections and Visual Checks of certain system parts are performed by a human inspector, often called a “tank diver”. Figure 2.10 shows an illustration of such a visual inspection. For the inspection, the human has to enter the tank through a manhole on the bottom side of the wing. Then, equipped with a flashlight and personal protective equipment, they have to manually inspect the tank. A second person has to assist to ensure the safety of the tank diver. Before starting the inspection, several safety precautions must be taken to remove fuel residue and explosive or toxic gases.

Tasks suitable for automation by robots are often described with four “D”s: Dirty, Dull, Dangerous, and Dear. Generally, “the more of these attributes a given task has, the more likely it is to be turned over to digital machines.” [MB17] The task of fuel tank maintenance fulfills all of them. It is dirty because of the fuel residue in the tank, it is dull because it is a repetitive task that takes a lot of time. The task is also dangerous because it presents health risks to the inspector. Finally, it is dear, i. e., expensive, referring to the high cost of operations and the critical nature of the inspection. All of these criteria make the task of fuel tank inspection relevant for automation using robots.

2.12 Endoscopic Robots

The objective of fuel tank inspection is similar to the idea of endoscopic robots in medicine because in both cases, a robot enters through a small hole into a larger cavity where

Figure 2.11: Da Vinci Xi, a robot used in robot-assisted minimally invasive surgery[10]

it has to move around. The type of surgery performed with these robots is called **robot-assisted minimally invasive surgery** (RMIS). In RMIS, the robot is usually composed of two parts, the proximal and the distal part. The proximal part is situated outside of the patient, often attached to a table or fixed frame. The distal part is inside the patient and contains further joints that move tools during the surgery. On the body of the patient, a trocar, a medical device which consists of a hollow tube, is attached and allows the distal part to pass through into the body cavity.

The link connecting the proximal and the distal part must always pass through the trocar, making its position a so-called **remote center of motion** (RCM). Some robots, for example, the DaVinci surgical robot, ensure this by the design of the robot [Che+20]. Such a structure may consist of parallelograms or belts that constrict the robot's movements. While mechanical enforcement of the constraint is generally considered safer, there is also an interest in software-based remote center of motion enforcement because it allows for more flexible use of robots and less elaborate mechanical constructions.

Figure 2.12 shows a surgical tool that should move through the trocar. The point denoted with p_{rcm} is the point that should match the trocar position. It can be defined as

$$p_{\text{rcm}} = p_{\text{pre}} + p_r^T \hat{p}_s \hat{p}_s$$

where p_{pre} is the beginning of the surgical tool that goes though the RCM, $\hat{p}_s$ is the unit vector in direction of the surgical tool link, and $p_r = p_{\text{trocar}} - p_{\text{pre}}$. The error, also called residual, i. e., the difference between the desired and actual value, is

$$r_{\text{rcm}} = p_e^T p_e = (p_{\text{trocar}} - p_{\text{rcm}})^T (p_{\text{trocar}} - p_{\text{rcm}}).$$

Deriving r_{rcm} with respect to q yields the Jacobian of the RCM J_{rcm}.

[10]https://www.intuitive.com/de-de/products-and-services/da-vinci, accessed 20/08/2024

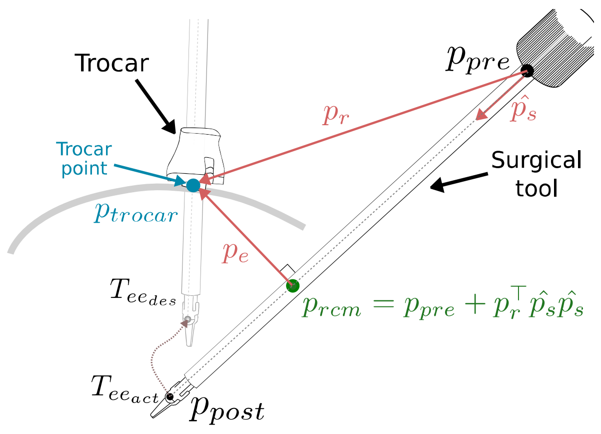


Figure 2.12: Remote center of motion in a surgical robot [DCH24]

$$J_{\mathrm{rcm}} = -2p_e^T \frac{\partial p_{\mathrm{rcm}}}{\partial q} = -2p_e^T \left((I_3 - \hat{p}_s \hat{p}_s^T) J_{\mathrm{pre}}(q) + (\hat{p}_s p_r^T + p_r^T \hat{p}_s I_3) \frac{\partial \hat{p}_s}{\partial q} \right)$$

with

$$\frac{\partial \hat{p}_s}{\partial q} = \frac{1}{\|p_s\|} (I_3 - \hat{p}_s \hat{p}_s^T) \big(J_{\mathrm{post}}(q) - J_{\mathrm{pre}}(q) \big).$$

A detailed derivation of this result can be found in Appendix A of [DCH24].

This Jacobian of the RCM describes how each joint before the RCM changes the position of the RCM. It fulfills a similar role for the RCM as the end effector Jacobian does for the end effector. Its usage for inverse kinematics solvers in medical contexts is explored in Sections 3.1.7 and 3.1.8. The application of the remote center of motion for fuel tank inspection is discussed in Section 8.5.

Chapter 3

Related Work

This chapter reviews related work, beginning with a presentation of inverse kinematics solvers that are relevant for the remainder of the thesis. It then presents different existing benchmarking approaches.

3.1 Inverse Kinematics Solvers

In this section, existing Inverse Kinematics solvers are described. The basic functionality of each solver is outlined and their strengths and weaknesses are discussed. Figure 3.1

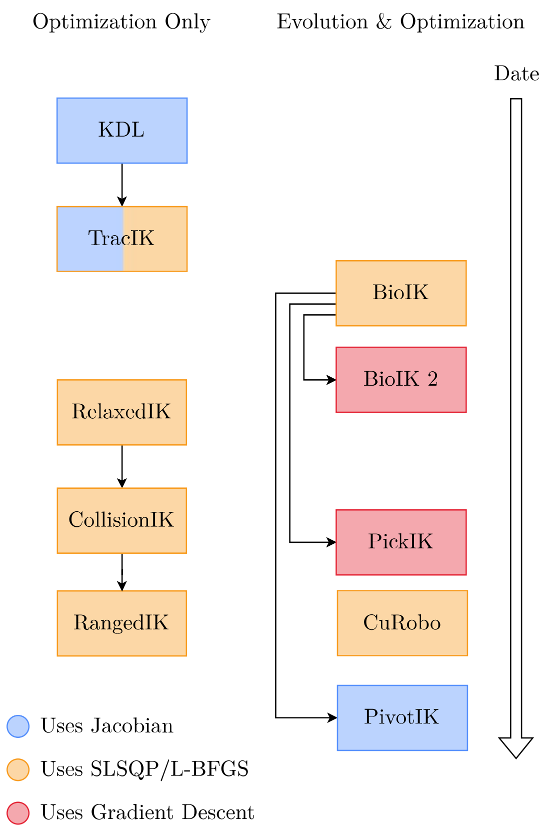


Figure 3.1: Family tree of IK solvers. Left is optimization-only, right combines evolution with optimization. The optimization methods are represented by colors. Publication date increases from top to bottom. Relations between solvers are marked with arrows.

shows an overview of the described solvers, including their underlying approaches. The solvers presented in this chapter form the basis for the improvements that are implemented in Chapter 8.

3.1.1 KDL

Arguably the most widespread inverse kinematics solver in the robotics community [BA15] is the Orocos KDL solver (Open Robot Control Software Kinematics and Dynamics Library) [SAO21]. It is the default inverse kinematics solver in MoveIt [CSC24]. The solver is a numerical solver that is based on the inverse Jacobian.

It works in the following way (see Algorithm 3.1): Starting with a given seed or all-zero configuration, forward kinematics are computed. Then, the difference of the end effector pose to the target pose is computed. The difference is computed as the position difference and the axis-angle representation of the rotation differences. From this, an error is computed as the maximum of the linear distance and the angular distance between the poses (linear and angular error). If both errors are below a threshold, the problem is considered solved. Otherwise, Newton's method is used to find a configuration where the joint configuration error is zero. Newton's method is a method that can find a root of a function by repeatedly calculating the root of the function's tangent at a candidate solution and taking this root as the new candidate solution.

In KDL, the root of the error function $e(q)$ has to be found, where the error function describes the distance of the end effector pose to the target pose when the robot is in configuration q. Using Newton's method with the function $e(q)$ gives $q_{k+1} = q_k - \frac{e(q_k)}{e'(q_k)}$.

```
KDL(target, timeout, seed):
 1  $q \leftarrow$ seed
 2  while timeout not exceeded
 3      ee_pose $\leftarrow$FK($q$)
 4      $\Delta$twist $\leftarrow$DIFF(ee_pose, target)
 5      if MAX($\|\Delta\text{twist}.p\|, \|\Delta\text{twist}.r\|) < 10^{-5}$:
 6          return true
 7      $J \leftarrow$JACOBIAN($q$)
 8      $J^\dagger \leftarrow$PSEUDO-INVERSE($J$)
 9      $q \leftarrow q - J^\dagger \Delta$twist
10      $q \leftarrow$CLIP($q$)
11  return false
```

```
DIFF($a$, $b$):
 1  return $a.p - b.p$, AXIS-ANGLE($a.r^{-1} b.r$)
```

Algorithm 3.1: KDL algorithm

The derivative of $e(q) = |f(q) - t|$ is the Jacobian matrix of $f(q)$ because the linear error term disappears in the derivative. Using the Jacobian, the step of Newton's method becomes $q_{k+1} = q_k - J^{-1}(q_k)e(q_k)$ where $J^{-1}(q_k)$ is the inverse of the Jacobian. Since the inverse cannot be calculated when the Jacobian is not square, the pseudo inverse $J^{\dagger}$, a generalization of the inverse for non-invertible matrices, is used instead. After the value of q_{k+1} is determined, forward kinematics can be calculated again and if the error is still larger than the tolerance, the next step of Newton's method is calculated to get a better approximation.

In 2019, a few changes have been made to the solver to avoid the solver being stuck and improve the handling of singularities[11]. A change to avoid singularities is to reduce the step size, i. e., apply only a fraction of the update in line 9 in the algorithm, when the error increases from one iteration to the next. When the solver is stuck in a singularity, so the solution is not reached but no update can be made, all joints are wiggled randomly to increase the chance of finding a new solution. In addition, joints that were clipped in line 10 of the algorithm are weighted lower in the next iteration. That means that their corresponding columns in the Jacobian matrix are weighted lower, leading to smaller updates for these joints.

While this approach is quite fast for simple scenarios and is practically unbeaten when a seed configuration close to the target configuration is given, Newton's method can get stuck in local minima. Additionally, it is not possible to include joint limits directly in the calculations because they are not represented in the error function. The only way to include joint limits is to clip the intermediate solutions, resulting in a worse approximation of a solution or possibly in no solution at all because the solver gets stuck. While the solver changes described above mitigate this partly, they remain a workaround. It is also not possible to include other constraints such as collision avoidance or arbitrary cost functions into Inverse Jacobian-based methods.

3.1.2 TracIK

An approach that tries to improve several disadvantages of KDL is TracIK [BA15]. TracIK starts two processes in parallel, one Inverse Jacobian-based and one optimization-based solver, and returns the solution of the process that finishes first. The Inverse Jacobian-based solver uses the normal KDL solver, but when the solution is not optimal and does not improve, i. e., a local minimum is detected, the solver is restarted with a random seed. For the optimization-based solver, the problem is formulated as a nonlinear optimization problem that is solved using sequential quadratic programming. The nonlinear optimization problem minimizes the objective function $\Phi = p_{\text{err}} \cdot p_{\text{err}}^T$ where p_{err} is the six-element Cartesian space error vector of the candidate solution. The authors also investigated using the L2-norm of the error vector and the dual quaternion distance between the solution and the target, where the latter in theory better integrates linear and rotational error, but both were outperformed by the dot product. In the problem definition given to the solver, constraints are added for the joint limits. The SLSQP (Sequential Least Squares Programming) algorithm was used, implemented in the nlopt-cpp library. Internally, it uses the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm which is a Newton-approximation method. Therefore, the random restart approach that was added

[11] https://github.com/moveit/moveit/pull/1321

to the Inverse Jacobian solver proved useful for the optimization solver as well. Since the SQP solver can natively handle constraints, an improvement regarding situations where the Jacobian-based solver runs into joint limits was achieved. The approach of formulating the IK problem as an optimization problem is also applied in other solvers such as the RelaxedIK family that is described in Section 3.1.6. TracIK also slightly changed the error metric used for termination:

```
TERMINATION(target, current):
1  $\mathrm{err}_{\mathrm{pos}} \leftarrow \mathrm{target.rot}^{-1} * (\mathrm{target.pos} - \mathrm{current.pos})$
2  $\mathrm{err}_{\mathrm{rot}} \leftarrow \mathrm{target.rot}^{-1} * \textsc{Axis-Angle}(\mathrm{target.rot}^{-1} * \mathrm{current.rot})$
3  return $\textsc{Equal}\big((\mathrm{err}_{\mathrm{pos}}, \mathrm{err}_{\mathrm{rot}}), 0, 10^{-5}\big)$
```

Algorithm 3.2: Termination metric used in TracIK

This metric uses the vector necessary to translate “target” to “current” and the rotation axis and angle to rotate “target” to “current”. To verify whether the difference is small enough, it is checked whether all of these values are approximately equal to zero, with a tolerance of 10^{-5}. Compared to KDL’s termination metric, the multiplication of target.rot^{-1} does not change the magnitude of the error vectors, only the reference frame. The equality check compared to the norm check introduces some additional freedom of a factor of $\sqrt{3} \approx 1.73$. The reason for the difference is two-fold. Previously, both solvers used the EQUAL method on the difference between “target” and “current”. TracIK added the multiplication by target.rot^{-1} in 2016 to make it possible to specify permissible error margins in the target frame[12]. KDL changed their metric in 2019 to unify the handling of position-only and full-pose-IK[13]. The practical difference between the metrics is considered negligible.

The TracIK solver provides different modes. The default mode is Speed, returning a solution as soon as it is found. Other modes are Distance, Manip1, and Manip2, where Distance reduces the joint space distance to the seed state and the Manip modes return the solution that had the highest manipulability. Manipulability means that the configuration is far away from a singularity. Manip1 aims to maximize the manipulability measure μ, Manip2 minimizes the condition number κ, as defined in Section 2.5. All modes except Speed use the complete available time to find as many solutions as possible to choose from. As joint space distance and manipulability are not considered in this thesis, only the Speed mode is used.

An advantage of this solver is that it is very fast and beats the KDL algorithm in all cases because it incorporates the algorithm into the solver. A disadvantage is that no additional constraints or cost functions can be specified.

3.1.3 BioIK

BioIK [SHZ19] is a memetic solver that combines evolutionary algorithms, particle swarm optimization, and gradient-based solvers. An evolutionary algorithm represents candidate solutions as individuals in a population, where good individuals reproduce to create off-

[12]Commit 6994051a2e1cdb45294ffa4883caa823879a7009
[13]Commit 6cda52babe508546f8d28b9abc08f33648d27ce1

spring by mixing their solutions. In BioIK, each individual is a configuration of the robot. In addition to the configuration, each individual also has a momentum that describes the difference to its parent. This element of particle swarm optimization is used to track improvements of the solution between generations.

The population starts with $n-1$ random configurations and the seed configuration. The individuals are then ranked by a fitness function that includes the distance of the end effector from its target pose but can also contain arbitrary other cost functions. In each step of the algorithm, the individuals reproduce and mutate, the best individuals are exploited and the worst ones are removed. For exploitation, the elite individuals, i. e., the k best individuals according to the fitness function, are optimized using the algorithm L-BFGS. These elite individuals cannot be removed from the population.

Then, all individuals in the population are added to a mating pool. They are the possible parents that reproduce to create offspring. For each of the $n-k$ non-elite individuals of the population, parents are selected from the mating pool and create a new individual by recombination, mutation, and adoption. The parent selection happens with a rank-weighted random function, i. e., the best individuals are more likely to be picked than the worst. In recombination, the genes (i. e., joint configurations) of the parents are mixed by applying a randomly weighted average of their genes and adding a random mix of their gradients. For mutation, a mutation rate is calculated that depends on the rank and fitness of the parents, so that children of less fit parents are mutated more than those of fitter parents. Finally, the created child is adopted by randomly adding genes from the average of their parents and a third individual called prototype selected in the same way as the parents. This adoption serves the purpose of adding a momentum to individuals that brings them in the direction of better individuals and is inspired by particle swarm optimization. The fitness of the new individual is then evaluated and if its parents are less fit, they are removed from the mating pool. This removal of parents is used to encourage niching so that multiple local extrema can be exploited in parallel.

The new individual or, if the mating pool is empty, randomly created individuals, are added to the offspring. Elites and offspring form the new population which is again evaluated for its fitness. If an individual is close enough to the target, it is returned as the solution. There is also a wipe-out criterion that detects if no improvement to the best

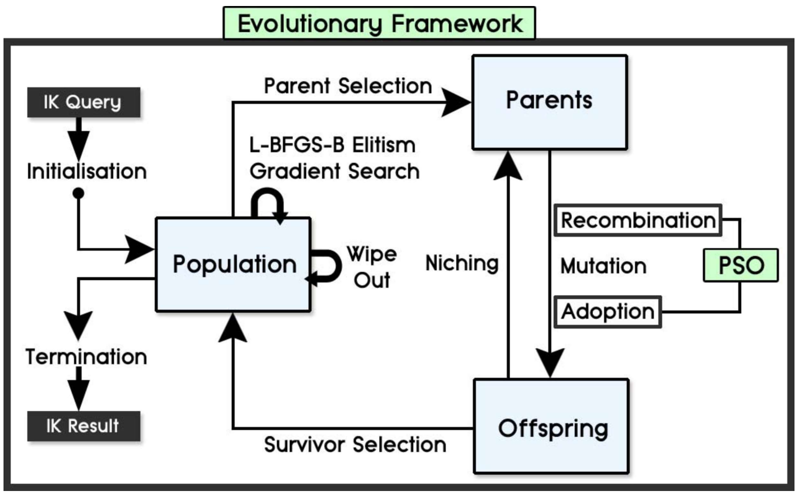


Figure 3.2: BioIK evolutionary algorithm [SHZ19]

```
BioIK():
1  assign seed as solution
2  initialize population
3      incorporate solution
4      create PopulationSize − 1 random individuals
5      evaluate and sort individuals by fitness
6      calculate extinction factors
7      try update solution
8  while not terminated do
9      assign whole population to mating pool
10     for all elite individuals do
11         perform exploitation using L-BFGS-B
12     for all non-elite individuals do
13         if mating pool is not empty then
14             select parents and prototype from mating pool
15             create new individual by recombination, mutation, and adoption
16             constrain genes to dimension bounds
17             evaluate fitness
18             remove worse parents from mating pool
19         else
20             create random individual
21             evaluate fitness
22         add individual to offspring
23     select elites and offspring as new population
24     sort individuals by fitness
25     calculate extinction factors
26     try update solution
27     if wipe out criterion fulfilled then
28         reinitialize population
29 return solution
```

Algorithm 3.3: BioIK algorithm from [SHZ19], slightly adapted

solutions was made in the last iteration. In this case, the whole population is reinitialized. The authors state that population size and number of elites are the only parameters of the algorithm, and they recommend between 50 and 150 individuals and 2-5 elites. Algorithm 3.3 shows an overview of the algorithm, Figure 14 illustrates its flow graphically.

BioIK supports the following goals: Position, Orientation, Look At, Distance, Displacement, Joint Value, and Joint Function. Position and Orientation are goals specifying the target pose directly. Look At rotates the end effector such that it is oriented toward a target. Displacement minimizes joint differences to the previous iteration, Joint Value minimizes the distance to a target joint value, and Joint Function minimizes an arbitrary function of the joint configuration. The Distance goal is used for collision avoidance. The authors acknowledge that “collision avoidance is an issue which is often neglected by generic inverse kinematics solvers” [SHZ19]. The error for the distance function is defined as

$$e_{\text{distance}} = \begin{cases} \infty \text{ if } d \leq d_{\text{min}} \\ \frac{1}{d-d_{\text{min}}} \text{ else} \end{cases}$$

where d is the distance to any object and d_{min} is a distance threshold. According to the paper, this goal is “fast to compute” but no more information on the computation of the distance itself is given. Since the algorithm is implemented for Unity, it is possible that collision object information is provided by the game engine.

The algorithm provides an interesting approach to solving inverse kinematics by combining multiple bio-inspired techniques. This encourages exploration of the solution space but is more time-intensive. The results presented in [Sta+16] show that BioIK has a higher solve rate than TracIK but requires at least 40 times as much time. The original implementation is only available for Unity and not open source.

3.1.4 BioIK 2

BioIK was reimplemented in C++ as a MoveIt plugin in [Rup17]. Using BioIK as a starting point, an adaption of the BioIK algorithm was also developed with a focus on performance increase. This algorithm, called BioIK 2 [Rup+18], differs in some ways from the original implementation.

First of all, the algorithm is run with multiple threads in parallel (default: 4), these are called islands. On each island, multiple species (default: 2) develop next to each other and compete for the best individual. The evolutionary and memetic optimizations are always executed for multiple generations instead of a single time. The population size and number of children are also selected differently, there are only two individuals in the population that are used to create 16 children. For these children, the so-called secondary goals are evaluated and used to pre-select a random number between 1 and 16 of them. Then, for the pre-selected children and the parents, their fitness according to the primary goals is evaluated and only the best two individuals are kept as the next parents. The differentiation between primary and secondary is done according to the computing expense of the goals; goals that can be evaluated in joint space and are therefore cheaper to evaluate are defined as secondary goals.

For the best individual in a species, exploitation is performed using a custom gradient descent method. For determining the step size used when the gradient is applied, a line search is implemented that can use either a linear or a quadratic equation to approximate the fitness landscape and estimate the optimal step size. A wipeout is only performed for the species on an island that does not contain the best individual and only if it did not improve in the last step or with a random probability of 10%. It replaces all individuals in the population with the same random individual. The algorithm terminates when a sufficiently good solution is found or a timeout occurs. The termination metric is the same as in TracIK for the PoseGoal. For additional goals, the result of their cost function must be less than 10^{-10}.

The implementation also provides a substantial number of pre-defined goals that can be combined and weighted to form the particle cost functions. These goals include a MaxDistanceGoal between a link and a target, a Line Goal, Direction Goal, Avoid Joint Limits Goal, an IKCostFnGoal accepting arbitrary MoveIt IK cost functions, and many more. All BioIK goals except the DistanceGoal usable for collision avoidance are also implemented, presumably the DistanceGoal was removed because of its high computational complexity.

BioIK 2 focuses on performance optimization compared to the original algorithm and other existing solvers. Most changes to the algorithm have been evaluated and shown to improve the efficiency in [Rup17]. A notable performance increase that is not algorithmic

```
BioIK2():
1  assign seed as solution
2  for all (2) species
3      initialize population
4          set all (2) individuals to solution
5  while not terminated do
6      for all species
7          for memetic_evolution_gens generations
8              for all children (16)
9                  select (the only 2) parents from population
10                 create new individual by mutating the first parent and adding
                   a mix of the parent gradients
11                 constrain genes to dimension bounds
12             sort children by secondary goals
13             pre-select a random number of children (1..16) by secondary
               goals
14             for all pre-selected children and parents
15                 evaluate primary fitness
16             keep the best (2) individuals in the population
17         for all elite individuals (1)
18             for memetic_opt_gens generations
19                 perform exploitation using gradient descent
20     sort species by fitness
21     if a species (except best) did not improve or w.p. 10%
22         wipe out this species
23     choose best individual as solution
24 return solution
```

Algorithm 3.4: BioIK 2 algorithm

is the usage of approximated forward kinematics. For that, forward kinematics is only calculated once per step (after line six in Algorithm 3.4) and only for the best individual. For all consecutive forward kinematics calculations, the solution is extrapolated using the Jacobian. This is done by multiplying the Jacobian with the joint offsets and adding the result to the calculated forward kinematics solution.

Mode	**Description**	**Threads**
bio2	BioIK 2 without optimization (purely evolutionary)	4
bio2_memetic	BioIK 2 with memetic optimization and quadratic step size (default)	4
bio2_memetic_l	BioIK 2 with memetic optimization and linear step size	4
bio2_memetic_lbfgs	BioIK 2 using LBFGS for memetic optimization	4
bio1	BioIK	4
gd	Gradient Descent	1
gd_2		2
gd_4		4
gd_8		8
gd_r	Gradient Descent with reset when stuck	1
gd_r_2		2
gd_r_4		4
gd_r_8		8
gd_c	Gradient Descent which accepts current solution when stuck	1
gd_c_2		2
gd_c_4		4
gd_c_8		8
jac	Pseudo-Inverse Jacobian Solver	1
jac_2		2
jac_4		4
jac_8		8
neural	Neural network predicting offsets to initial solution	1
neural2	Neural network predicting solution	1
optlib_*	Optimization solvers using BFGS/L-BFGS-B/L-BFGS/NelderMead/Gradient Descent/Conjugated Gradient Descent/Newton Descent	1/2/4

Table 3.1: BioIK modes

The C++ implementation of BioIK supports various modes shown in Table 3.1. The bio1 algorithm implements the original BioIK algorithm, with a change in the exploitation method. Instead of using L-BFGS, a heuristic exploitation method is used that iterates over all genes, randomly increases and decreases them, and keeps changes that improve the result. The bio2 algorithms are four variants of the optimized algorithm. They differ in their use of exploitation of the elite solutions. When used, it can either be calculated using gradient descent with linear or quadratic step size, or with a numerical solver (L-BFGS, as in the original BioIK algorithm). Since the paper found that the quadratic step size for gradient descent performed best, this is the default solver. In addition, a gradient descent solver with different behaviors when the solver is stuck on a local minimum and a pseudo-inverse Jacobian solver is implemented. These solvers are available with a different number of threads (one to eight). Another family of solvers is added that is purely optimization-based. These solvers are implemented using cppoptlib and can use different numerical solvers. Two implementations based on neural networks are also implemented, one that predicts the solution to the IK problem and one that predicts offsets from the current configuration required to achieve the target pose. Both of them are trained during the startup of the solver.

An advantage of the BioIK C++ package is that it provides many different solvers that can be compared to each other. Also, the variety of available goals and the option of adding custom goals makes the solver applicable to many different situations. On the other hand, the code is research code that is hard to read and therefore hard to adapt, in many cases it is unclear which parts of the code are actually used, and an overview of the algorithm (created in Algorithm 3.4 as part of this thesis) is missing. In addition, it is not clear whether the default “bio2_memetic” algorithm is always the best or if there are some use cases where other solvers are better.

3.1.5 PickIK

PickIK [Wea24] is a reimplementation of BioIK that focuses on code quality and thread safety. PickIK is implemented by researchers from PickNik Robotics, a robotics company that is using and supporting ROS 2 and MoveIt. PickIK is open source, as are all of the solvers mentioned, and was until recently actively developed. During the writing of this thesis, PickIK maintainers stated that they no longer work on the IK solver and switched to TracIK[14].

PickIK offers more parameters for the evolutionary algorithm, for example population and elite sizes, wipeout tolerance, and maximum number of generations. A main disadvantage of PickIK, especially compared to BioIK 2, is its low speed. According to profiling results, this speed difference can mainly be explained by the forward kinematics that are extrapolated in BioIK 2 but fully calculated in PickIK. The authors are aware of the issue but aimed to incorporate the method into the larger MoveIt framework rather than implementing it in their own code.[15] This would have the advantage that it would be available to more users but also takes more time as more people are involved in the required decisions.

[14]https://github.com/PickNikRobotics/pick_ik/issues/70
[15]https://github.com/PickNikRobotics/pick_ik/issues/60

PickIK also implements different modes. The default mode is "global", in this mode, the evolutionary algorithm with the memetic optimization is used. The "local" mode is an exact reimplementation of the "gd" algorithm of BioIK 2, i. e., a gradient descent solver with linear step size estimation.

An extensive code review of PickIK shows multiple differences in implementation to the BioIK algorithm. As no research paper has been published for PickIK, it is not known whether these differences are done on purpose or if they are implementation errors, but the authors claim to have an implementation "equivalent to `bio1` and `bio2_memetic` solvers". In reality, they present a solution that is similar to `bio1`, but has the following deviations from the original algorithm:

- For exploitation, gradient descent with linear step size estimation (as in `bio2_memetic_l`) is done instead of using L-BFGS-B.
- During reproduction, only elites are used as parents and they are selected uniformly. Originally, all individuals are considered parents, and their choice is weighted by their rank.
- After recombination, the gradient of the child is set to zero instead of a random mixture of the parent gradients.
- No adoption is performed.
- In the wipeout check, only the best solution is considered when the iteration is checked for improvement instead of all elites.
- Wipeout completely resets the population instead of keeping the best solution.
- The termination metric of KDL (linear and angular distance) is used, but with a default threshold of 10^{-3} instead of 10^{-5}.

Especially the last point might be crucial to the algorithm's performance, as the solver essentially forgets all previously made progress whenever a wipeout occurs.

3.1.6 RelaxedIK

Another solver family is the RelaxedIK family, currently consisting of RelaxedIK, CollisionIK, and RangedIK. All solvers in this family are optimization-based solvers that formulate the problem as a constrained non-linear optimization problem. The objective function is a weighted sum of goal functions that differ between the solvers of the family.

The first version, RelaxedIK [RMG18], lays its focus on generating smooth trajectories. For that, in addition to position and orientation goals for the end effector and a function for self-collision, goals are added that minimize the velocity, acceleration, and jerk of the joints and the velocity of the end effector. For all of these goals, a special objective function, called groove loss, is used. This groove loss is defined as the sum of a polynomic function, in this case of degree four, and a Gaussian. This makes it possible to have the loss function generally follow a polynomic, making it easy to have a clear gradient leading to the target value, and have a "groove" around the target value (Figure 3.3 D).

$$\text{groove}(x) = (-1)^n \exp\left(-\frac{(x-s)^2}{2c^2}\right) + r \cdot (x-s)^g$$

In the groove loss function, x is the parameter that is the optimization target, it is the output of a goal function. The other parameters determine the shape of the loss function.

s corresponds to the center of the function, c gives the width of the Gaussian, r changes the transition between the polynomial and the Gaussian, g is the degree of the polynomial, and $n \in \{0, 1\}$ changes the sign of the Gaussian, either punishing ($n = 0$) or rewarding ($n = 1$) values of x near s.

The authors claim that this function makes it easier and more intuitive to choose weights for different objectives because the output for any input value to the loss function is -1 for the optimal value, 0 at the edge of the groove, and then increases to infinity. In this way, multiple objectives can be combined using a weighted sum of their losses. The groove in the loss function makes it possible to fulfill conflicting goals by deviating from some goals to fulfill others, for example, it is possible to slightly deviate from the target orientation if this helps to avoid a high jerk in a joint.

In addition to the objective function, hard constraints are added for joint position and velocity limits as well as the manipulability measure μ which is used to steer clear of singularities. An interesting approach for self-collision is that a neural network is used for the detection. According to the authors, the neural network trains comparably fast (30 minutes) and performs the check faster than conventional methods.

To solve the optimization problem, scipy's SLSQP solver is used. With this approach, the state-of-the-art solver TracIK could be outperformed by the authors on the measure of trajectory smoothness, but not on the measure of end effector pose error.

CollisionIK [Rak+21], the next iteration of RelaxedIK, adds environment collision avoidance to the solver. In addition to the goals of RelaxedIK, a goal for collision avoidance is used. This goal has the form $\frac{\varepsilon^2}{d^2}$ which is calculated for the distances d between all links and collision objects and then summed up. The resulting value is used as input to the groove loss function. ε is two times the link radius, which was chosen as a cut-off distance between collision and non-collision. The distance checking uses the ncollide3d [Cro22]

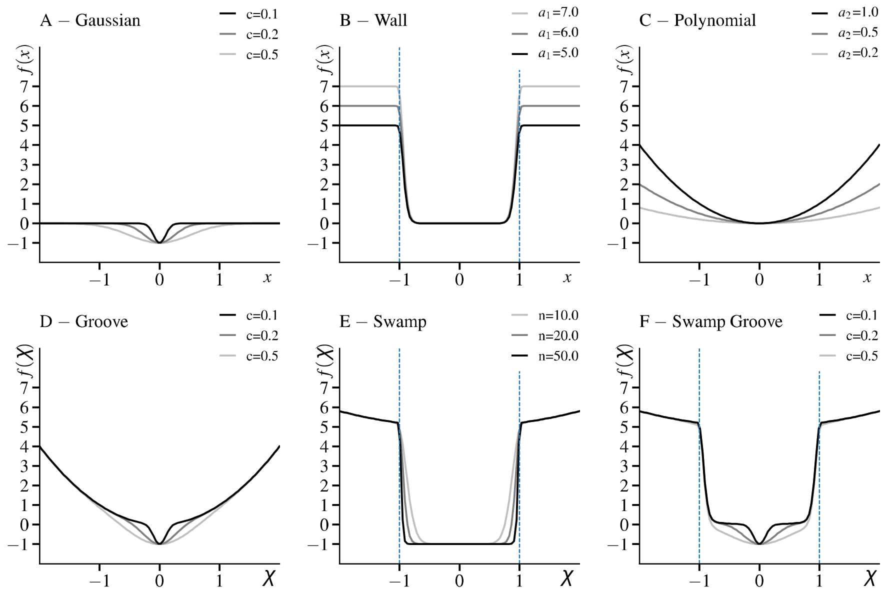


Figure 3.3: Loss functions used in the RelaxedIK family. Top row shows building blocks for bottom row. [Wan+23]

library for collision checking on the convex hulls of objects. It uses axis-aligned bounding boxes in a Bounding Volume Tree in the broad phase and the GJK algorithm in the narrow phase.

CollisionIK supports different modes: In addition to the "normal" mode, the orientation goal can be completely ignored, making it possible to better plan collision-free trajectories where the end effector *position* follows a trajectory, and the adaptive orientation mode, where the weight of the orientation goal is adapted based on how close the robot is to collision, keeping the requested orientation when possible but adapting its weight when it is necessary to remain collision-free.

In this solver, the optimization method was changed to the proximal averaged Newton-type method (PANOC), but the authors state that it also works with other non-linear solvers such as SLSQP.

The newest version of the solver, RangedIK [Wan+23], adds a new loss function, the swamp loss (Figure 3.3 E). This function has a range where it has a low value bounded by "walls" that drive the objective away from too extreme values, giving the variable a valid range of values. Semantically, it represents goals where a range of values is good for the objective, for example to avoid joint limits.

$$\mathrm{swamp}(x') = (a_1 + a_2 x'^m)\left(1 - \exp\left(-\frac{x'^n}{b^n}\right)\right) - 1$$

$$x' = \frac{2x - l - u}{u - l}$$

x' scales x so that the range between lower bound l and upper bound u is mapped to $(-1, 1)$. The parameters a_1 and a_2 define the transition between the wall and the polynomial, n determines the wall steepness, b the wall locations depending on l and u, m is the degree of the polynomial. b can be determined by solving $1 - \exp(-\frac{1}{b^n}) = 0.95$ to ensure that a maximum penalty of 0.95 is given inside of the swamp.

The swamp loss can also be combined with the groove loss to form the swamp groove loss (Figure 3.3 F). This loss allows a goal where a range of values is desired but a certain value is preferred. For example, the end effector of the robot can be preferred to have a certain orientation but other orientations within a range around the goal are also valid. Figure 3.3 A-C shows the three building blocks gaussian, wall, and polynomial. Using them, the groove, swamp, and swamp groove loss functions can be built (Figure 3.3 D-F).

In addition, RangedIK drops support for environment collisions and changes the self-collisions from a neural network to a calculated method that uses the swamp loss to keep the distance between all non-adjacent links above a lower limit of two centimeters. A wrapper for RangedIK is also available for ROS 2, but only supports a limited message interface and is not integrated into a larger framework such as MoveIt [Rak24].

3.1.7 Inverse Kinematics of Minimally Invasive Surgical Robots

The research paper *A Concurrent Framework for Constrained Inverse Kinematics of Minimally Invasive Surgical Robots* [Col+23] compares different types of inverse kinematic solvers and their combinations for use in minimally invasive surgery. They compare an inverse-Jacobian method, an optimization method, and a hierarchical programming approach as well as concurrent combinations of them. All of the solvers do not only solve for the end effector pose but also for keeping the remote center of motion stable. For the inverse-Jacobian method, this is done in the following way: In addition to the usual term that uses the pseudo-inverse Jacobian for the desired end effector position, the Jacobian of the remote center of motion is considered. This is done by defining a hierarchy of the tasks and solving the second task in the null space of the first. Recall that the null space is the space of solutions that do not change any solutions for a particular task, for example, for a seven-degree-of-freedom robot solving for a six-dimensional pose, the redundant joint makes it possible to reach the target in multiple ways. This null space can then be used to optimize a second task. The implementation uses the following formula from Chiaverini [Chi97] that makes it possible to set different priorities for the tasks:

$$\dot{q} = J_{\mathrm{rcm}}^{\dagger} K_{\mathrm{rcm}} e_{\mathrm{rcm}} + \left(I_n - J_{\mathrm{rcm}}^{\dagger} J_{\mathrm{rcm}}\right) J_{\mathrm{ee}}^{\dagger} K_{\mathrm{ee}} e_{\mathrm{ee}}$$

K_{ee} and K_{rcm} are gain matrices that are set to $100I$ and I, respectively, where I is the identity matrix. J_{rcm} and J_{ee} are the Jacobians of the end effector and the RCM, as defined in Section 2.5 and Section 2.12, respectively. e_{rcm} is the distance between the shaft and the RCM target (p_e in Figure 2.12), e_{ee} is defined as the $\log_6(T_{\mathrm{err}})$ where T_{err} is the transform from the actual to the desired pose. $\log_6$ is the logarithm from Lie algebra, transforming an SE(3) transform to a six-dimensional vector. The resulting inverse-Jacobian based solver leverages both Jacobians to solves both tasks simultaneously.

In the solver using the optimization method, the tasks are weighted with a high weight for the RCM task and a low weight for the tool pose task. Then, their combined error is minimized using a nonlinear optimization solver, and constraints are added for joint limits. The optimization is done using IPOPT's HSL/MA57 solver.

The hierarchical programming approach formulates both tasks as quadratic problems with linear constraints. One task, in this case the RCM, is solved first, and its solution is incorporated into the formulation of the second task. This ensures that the solution of the second task does not violate the first solution.

Their evaluation shows that the best results were obtained by running the inverse-Jacobian solver and one of the other solvers in parallel, with a slight advantage for the hierarchical programming approach. This highlights that concurrently running multiple different solvers can leverage the advantages of both.

This paper uses the same approach as TracIK by running multiple different solvers in parallel. Incorporating the pivot point into the Jacobian method is a promising approach that can greatly improve the convergence speed.

3.1.8 PivotIK

PivotIK [DCH24] is a memetic algorithm by the same authors as the concurrent framework from the preceding section. Even though the authors do not explicitly say so, the algorithm is almost the same as the BioIK algorithm. Two changes have been made to include a remote center of motion.

First, the fitness function that is used to evaluate the individuals uses not only the end effector position error but also the distance to the remote center of motion. Different fitness functions were evaluated and even though the one used in BioIK resulted in the fewest number of generations, their fitness function was the fastest overall. The resulting function is

$$\Phi(x) = \mu_{\text{rcm}} e_{\text{rcm}}(x) + \mu_{\text{ee}} e_{\text{ee}}(x)$$
$$e_{\text{ee}(x)} = \| \log_6\left(T_{\text{ee}_{\text{des}}} T_{\text{ee}_{\text{act}}}^{-1}\right) \|_2$$

It is a weighted sum of the distance between the shaft and the RCM point (e_{rcm}) and the end effector error (e_{ee}). The end effector error is computed from the difference between the actual and desired poses using the $\log_6$ function from Lie algebra.

The second change is that the elite exploitation does not use a gradient descent approach because gradient computation is deemed too expensive by the authors. Instead, a single step of the inverse-Jacobian approach from their previous paper was used to optimize these individuals. The authors claim that the original gradient calculation from BioIK can cause convergence failures when more constraints are used.

Even though not explicitly stated in the paper, the algorithm published in their GitHub repository[16] also contains the particle swarm optimization concepts of BioIK, including gradients for each individual and the concept of adoption. The evaluations in the paper were done using the C++ implementation of the solver; this solver is not yet made open source, only a slower version written in Python was published. Also, PivotIK is compared to multiple other solvers but not to the solver the authors found was best in their previous publication.

3.1.9 CuRobo

Recently, NVidia published a solver that leverages GPU parallelization, called CuRobo [Sun+23]. It performs multiple IK queries with different seeds in parallel using an adapted version of L-BFGS and particle-based optimization. In this algorithm, the MPPI solver [WAT15], a highly parallel, sampling-based particle optimization solver is run for a few iterations. Its result is then used as a seed for L-BFGS, which can then further improve the precision. Both of these solvers are implemented as CUDA kernels and run in parallel on the GPU. The speedup of this method compared to other approaches is substantial but depends on the batch size. With a batch size of 100, it is 42x faster than TracIK when collisions are ignored and 85x faster when collisions are avoided. Larger batch sizes increase this difference. For small batch sizes or single-problem execution, the performance is generally worse than that of other IK solvers.

For collision detection, the robot is represented by a collection of spheres. The collision detection is split into self-collision avoidance and world collision avoidance. For self-collision, the penetration depth between pairs of spheres is calculated. The authors found that they could reduce the number of penetration depth tests by 50% by excluding pairs of spheres that could never collide. The self collision cost was then the maximum penetration depth of any two spheres. For world collision, a cost function is used that scales linearly to the penetration depth in the case of collision and was zero when the objects were not in collision, except for a buffer zone of three centimeters, where a quadratic function was used to have a smooth transition between collision and non-collision. The collision environment used for distance calculations of meshes uses a bounding volume

[16]https://github.com/davilac/PivotIK

hierarchy provided by NVIDIA warp. The speed of distance queries using this environment outperforms the existing methods PyBullet [CB16] and STORM [Bha+21] starting from a batch size of 1, at higher batch sizes the performance of CuRobo is several orders of magnitude better. [Sun+23]

The CuRobo solver is very promising because it makes use of GPUs for inverse kinematics and motion planning on a large scale. However, the project is still very new and not well-documented. At the time of writing, a MoveIt integration is available for the motion planner but not for the inverse kinematics solver[17].

3.2 Benchmarking

Benchmarking can be used to compare different solutions to a problem in a repeatable and reliable way. In this section, existing benchmark approaches are described. First, benchmarking techniques for inverse kinematics solvers are detailed, then a library for benchmarking of kinematic structures, REACH, is presented.

3.2.1 IK Benchmarking

Currently, there is limited research on benchmarking datasets for inverse kinematics. Benchmarking tools usually work in the following way: Random joint positions are sampled, forward kinematics is performed to calculate the end effector pose, and this pose is then given to the inverse kinematics solver. Examples of this approach can be found in [BA15, Col+23, Rup+18, SHZ19]. It ensures that all end effector poses are solvable and guarantees a certain configuration distribution in joint space. The joint configuration can also be sampled from a Halton sequence to ensure equal distribution across the joint space [Sun+23]. An example of a recently published library implementing this type of benchmarking is ik_benchmarking [Rae23]. While this type of benchmarking is easy to perform and can evaluate an algorithm for its speed and accuracy for simple inverse kinematics tasks, it fails to grasp more complex scenarios because neither is the actual workspace of the robot considered nor are environmental conditions factored into the result. An approach that improves on the first problem is to generate a grid of target poses that should be reachable by the robot and test which poses can be solved by the inverse kinematics solver. Still, the environment is not considered but the result is applicable to the (Cartesian-space) workspace area of the robot.

In research focused on collision checking, another approach is sometimes used. A task is created where the robot's end effector has to follow a certain Cartesian-space trajectory given some constraints, for example the end effector has to follow a square path while avoiding collision with a block. This approach is used in [Col+23, Rak+21, Sun+23]. [Sun+23] uses the MotionBenchMaker dataset [Cha+22], a benchmark for motion planning tasks. This dataset includes a number of different scenes and robots where tasks such as reaching for a certain object have to be performed. Solving such a task includes motion planning and collision-aware inverse kinematics. A theoretical advantage of the MotionBenchMaker is that it is easily extensible and multiple different tasks for a scene can be generated automatically by varying object positions and orientations. The dataset generator, the visualization, and the benchmarking software as well as the underlying

[17]https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_cumotion, accessed 22/08/2024

Solver	**Approach**		**Metric**			
	Joint space	Cartesian space	Solve rate	Average Error	Average Time	Others
TracIK [BA15]	✓	×	✓	✓	×	×
BioIK [SHZ19]	✓	×	✓	×	✓	×
BioIK 2 [Rup+18, Rup17]	✓	✓	✓	×	✓	×
RelaxedIK [RMG18]	×	✓	×	✓	✓	✓
CollisionIK [Rak+21]	×	✓	×	✓	×	✓
RangedIK [Wan+23]	×	✓	✓	✓	×	✓
Concurrent Framework [Col+23]	×	✓	✓	×	✓	×
PivotIK [DCH24]	✓	✓	✓	✓	✓	×
CuRobo [Sun+23]	✓	✓	×	×	✓	×

Table 3.2: Evaluation metrics of inverse kinematics papers

Robowflex library [KK22] by the same authors only support ROS 1 with no plans for ROS 2 support in the "immediate future".[18] This effectively makes extending the dataset and benchmarking ROS 2 software unfeasible with the MotionBenchMaker.

In Table 3.2, benchmarking approaches and metrics used in different papers on inverse kinematics solvers are aggregated. The approach refers to whether joint configurations were randomly generated or Cartesian-space goals were used. The most important metrics solve rate, average error, and average time were analyzed. Some papers included average rotational and positional error, this was combined as average error. Other metrics focus on goals such as trajectory smoothness or manipulability measure.

From the table, it is clear that Cartesian-space evaluation is both important and present in a lot of solvers. However, these evaluations are often limited to a specific use case. For example, [Col+23] and [DCH24] focus on keeping a remote center of motion; their evaluations therefore include a robotic arm following a trajectory while passing through a given point (see Figure 3.4). The most generalized evaluation approaches are present in CuRobo [Sun+23] and the RelaxedIK family ([RMG18], [Rak+21], [Wan+23]). For

[18]https://github.com/KavrakiLab/robowflex/issues/306

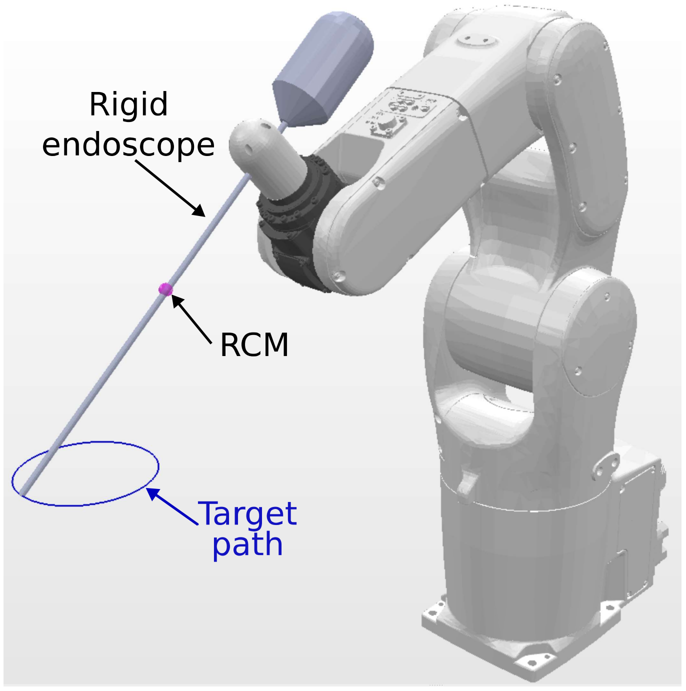


Figure 3.4: Example benchmark of inverse kinematics with a remote center of motion (RCM) [Col+23]

the RelaxedIK family, the code used for evaluation, however, is not published. CuRobo uses the MotionBenchMaker dataset, but for inverse kinematics evaluation, only a single instance of a single scenario (bookshelf-small-panda) was used. BioIK 2 benchmarks using a grid test and two wheel turning tasks, but no quantitative data is shown for the latter.

3.2.2 Kinematic Benchmarking

During the development of new robots, it is important to verify that a newly designed robot can adequately fulfill its task. In contrast to research applications where a robot has to fulfill several different purposes, industrial robots are often constructed for a specific use case and can therefore be adapted to better suit their task. To evaluate the kinematic structure of such a robot, kinematic benchmarking can be used. This type of benchmarking can be performed using the REACH (Robotic Evaluation And Comparison Heuristic) library developed by the Southwest Research Institute (SwRI) for their use in industrial applications. To evaluate the usability of the robot, a 3D model of an object is loaded into the library. The library then evaluates how well the robot can reach the object, which is relevant for industrial tasks like painting, soldering or inspecting the object. For this evaluation, points are sampled from the surface of the object and inverse kinematics is performed for each of the points and the given robot. Additionally, the resulting positions are evaluated for different goals, such as manipulability or distance to the joint limits. Based on these evaluations, a score is calculated and the target points are colored according to the score in a visualization to show how well the robot can reach the object. After the initial execution, an optimization loop is started that seeds the IK with results from nearby points to obtain solutions where none were found previously or improve previous results. The developers used the library to evaluate different kinematic structures for a robot that should remove the paint from an aircraft (see Figure 3.5).

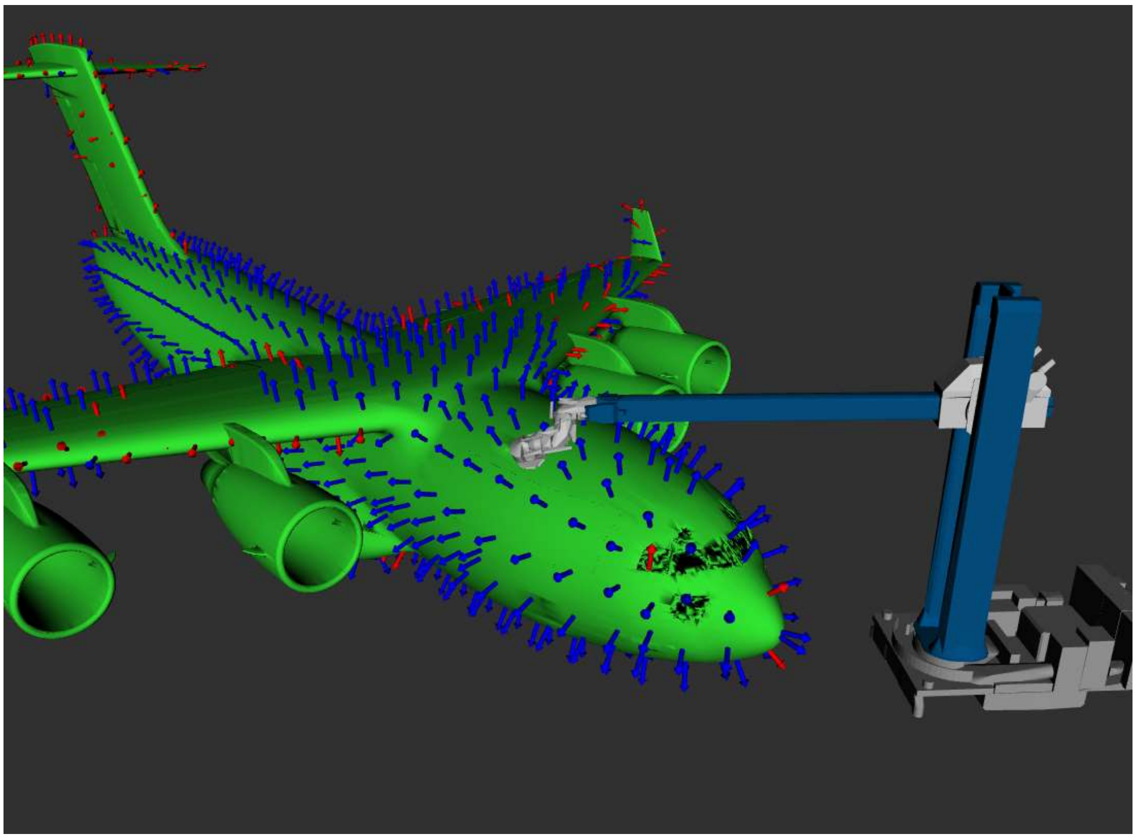


Figure 3.5: Kinematic benchmarking of a robot operating on a plane [Rip23]

While the development goal of the library was to evaluate kinematic structures, it can also be used to evaluate inverse kinematics. When the target points on the object are known to be reachable by the robot, the results of the library evaluate the performance of the inverse kinematics solver. Swapping the solver or changing its parameters cause a change in the reachability results that are captured by the library.

Additionally, the structure of the library is highly flexible and extensible. The inverse kinematics solver, the target point generator, and the evaluator are plugins that can be swapped out or modified to suit one's needs.

Chapter 4

Benchmarking Dataset

In this thesis, a benchmarking dataset for inverse kinematics is created to better compare different inverse kinematics solvers in endoscopic or high-collision scenarios. For this dataset, it has to be considered which tasks should be covered by the dataset and which kinds of scenarios should be created to represent the problem domain. For the tasks, the decision was made to add objects to the workspace of the robot and let the end effector touch random points on their surfaces. These surface points are then used as targets for the inverse kinematics solver, the object serves as a collision object.

Selecting single points as targets serves two purposes. First, a planner in robotics works from an initial (usually the current) configuration of the robot and the target configuration at the goal pose. Using these two poses, the planner works out a trajectory, for example by interpolating in the configuration space or by creating a Cartesian-space trajectory with waypoints. Regardless of the trajectory generation method, a joint configuration for the target pose has to be found. If no such target configuration is available, planning fails. Therefore, an IK solver used in combination with a planner must be able to solve any possible target position without being provided with a solution in proximity. The second reason for using single target points is the focus on inspection tasks. In these tasks, a planner generates points at or around an object that provide a high coverage of the object, i. e. all target surfaces of the object can be seen from the union of all viewpoints. This process is called coverage path planning and has been covered by extensive research [GC13]. After generating target poses, the robot's sensor has to travel along these points to collect the requested data. Once again, inverse kinematics has to be used for all of the points on the trajectory. A common approach is to use random sampling for point generation. The use case of coverage path planning can therefore also be covered by sampling random points on object surfaces.

For the dataset, seven objects were selected that provide varying levels of collisions. An additional scenario with no objects and random poses is also available. Figure 4.1 shows the objects in the dataset. The green object is the collision object, the red arrows are the target poses given to the solver.

The *Random* scenario was added to compare the behavior with and without collisions and to compare the benchmarking to previous, joint-based benchmarks. There are no collision objects in this scenario, only self-collisions are not allowed. The first two objects are two tables of different sizes, *Table* and *SmallTable*. Since tables are common everyday objects often used with robots, for example in research, industry, or household applications, the solvers are expected to perform quite well here. The target points are all sampled from

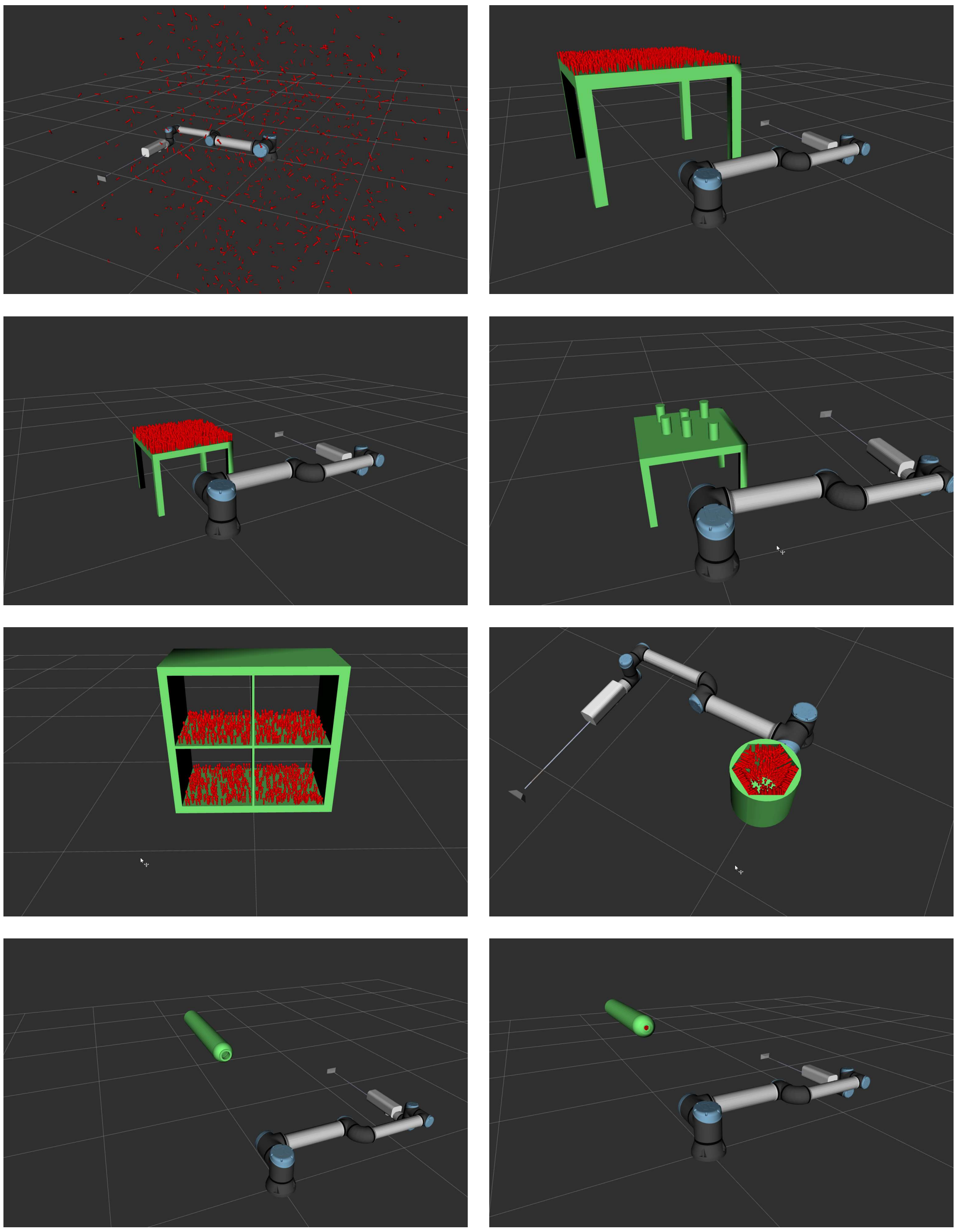


Figure 4.1: Scenarios in the evaluation dataset. In reading order: *Random*, *Table*, *SmallTable*, *TableObjects*, *Shelf*, *Barrel*, *HydrogenTank*, *HydrogenTankSmall*. The visualization is done in rViz.

the top surface of the table, and a low number of collisions is expected when trying to find a solution for these points. In the next scenario, *TableObjects*, six cylindrical objects are placed on top of the small table. Instead of the table surface, all reachable cylinder surfaces, i. e., the sides and caps, are used as targets. In Figure 4.1, the arrows are omitted to provide a clearer view of the objects. Again, this object was selected because it is a scenario relevant for research, industry, and household applications where an object should be grasped or inspected while being surrounded by multiple other objects. The next object is another common household object, a quadratic *Shelf*. It offers a higher number of possible collisions than the previous scenarios and still has a relevant use case, for example in pick-and-place or cleaning tasks. The structure of an individual section already provides some similarity to an endoscopic task as the targets are surrounded by obstacles on most sides. An even more constrained scenario is the next, the *Barrel* scenario, where the robot has to reach inside a barrel-like object standing on the ground. The inside of the barrel has been chosen to be hexagonal rather than round to avoid issues of the concave surface making it impossible to touch a target positions. The next two tasks represent hydrogen fuel tanks and are therefore called *HydrogenTank* and *HydrogenTankSmall*. The tanks are identical except their opening which has a diameter of 7 cm for *HydrogenTank* and 3.5 cm for *HydrogenTankSmall*. All target poses are on the inside of the tanks. These scenarios are particularly challenging due to the high number of collisions, representing real-world constraints in endoscopic inspection tasks.

All mentioned objects have been created in Blender and were exported as Stanford PLY files. The objects were then modified, removing all surfaces except the surfaces from which target points for the robot were sampled, i. e. the upper side of the table or the inner side of a fuel tank. These files are used to create PCD files by randomly sampling points on the surface of the object using `pcl_mesh_sampling` from the Point Cloud Library (PCL) package. The PCD files and the original PLY files are used as input for the REACH library for evaluating different inverse kinematics algorithms. It is important to note that even though the arrows suggest that the orientation of the end effector is free around the arrow's axis, due to limitations in REACH and MoveIt, a fully constrained target pose is given to the solver.

All objects and target poses were manually reviewed to make sure that all target poses are reachable with ELISE. Even though this is a manual overhead, it helps to make the results comparable and avoids differentiating between unreachable and unsuccessfully solved poses.

The final dataset containing the different scenarios suitable for evaluation solver performance in endoscopic and non-endoscopic environments with varying degrees of collisions is used in the remaining evaluations of this thesis.

Chapter 5

EBIKE

Since no evaluation library for inverse kinematics solvers in complex environments currently exists, one was created as part of this Master's thesis. This chapter outlines the underlying considerations, describes the implementation, and presents an example demonstrating the library's usefulness.

5.1 Motivation

To perform the evaluation of different IK solvers and the dataset, software was required that facilitates evaluation, comparison, and parameter optimization of the different solvers. Such a library should be able to create a baseline for inverse kinematics solvers and compare solvers to others regarding their success rate and speed in different scenarios. Development of new inverse kinematics solvers will be facilitated because changes to the solver can be evaluated more easily. With the results showing a trade-off between success rate and solving speed, the end user of the IK solver can decide which of these is more important as this depends on the problem domain (e. g., real-time vs. pre-planned). Components for persisting, visualizing, and printing results should also exist.

In addition to evaluation and comparison of different solvers and solvers with different parameters, there should also be structures that can be used for automatic parameter optimization of the inverse kinematics solvers. This makes hand-tuning parameters for specific situations dispensable because the situations and possible parameter ranges can be added to the library and automatic parameter optimization takes care of finding the optimal parameter values.

5.2 Implementation

In order to fulfill these requirements, the EBIKE library was developed as part of this thesis. EBIKE stands for Enhancement and Benchmarking of Inverse Kinematics in Environments.

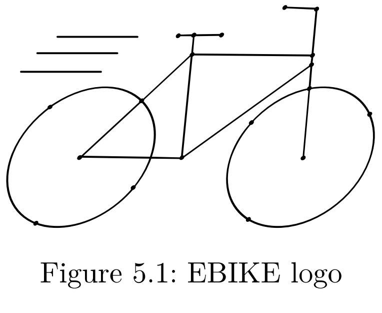


Figure 5.1: EBIKE logo

This library allows benchmarking and comparing any IK solvers that are implemented as MoveIt plugins. Parameter optimization is currently only possible for PickIK because it is the only available solver with parameters that can be adapted. The optimization uses the Optuna [Aki+19] library, which is typically used for hyperparameter optimization in machine learning applications. Given one or multiple objectives and multiple parameters, it aims to find the parameter set that minimizes or maximizes the objectives. In contrast to other optimization routines, hyperparameter optimization focuses on a low number of parameter evaluations and different types of parameter distributions, such as discrete, uniform-continuous, or logarithmic-continuous. The library uses the Tree-Structured Parzen Estimator (TPE) [Ber+11] for sampling of new parameters. The estimator chooses new sampling points by repeatedly determining the parameter values that yield the highest expected improvement. This is done by modeling the objective with a surrogate function that is optimized in its stead. The parameter values that maximize the surrogate are used as the new sampling point for the objective, the objective's result then updates the surrogate for the next iteration. Optuna is easy to set up and supports running on a distributed system.

The EBIKE library provides several base classes that can be extended, facilitating inclusion of custom scenarios, IK solvers, and robots. This helps to make the library future-proof, making it possible to use it for new problems.

Figure 5.2 shows the structure of the library as a UML class diagram. The main class is `Benchmark`. It uses REACH's Python API to run a REACH study on the given scenarios

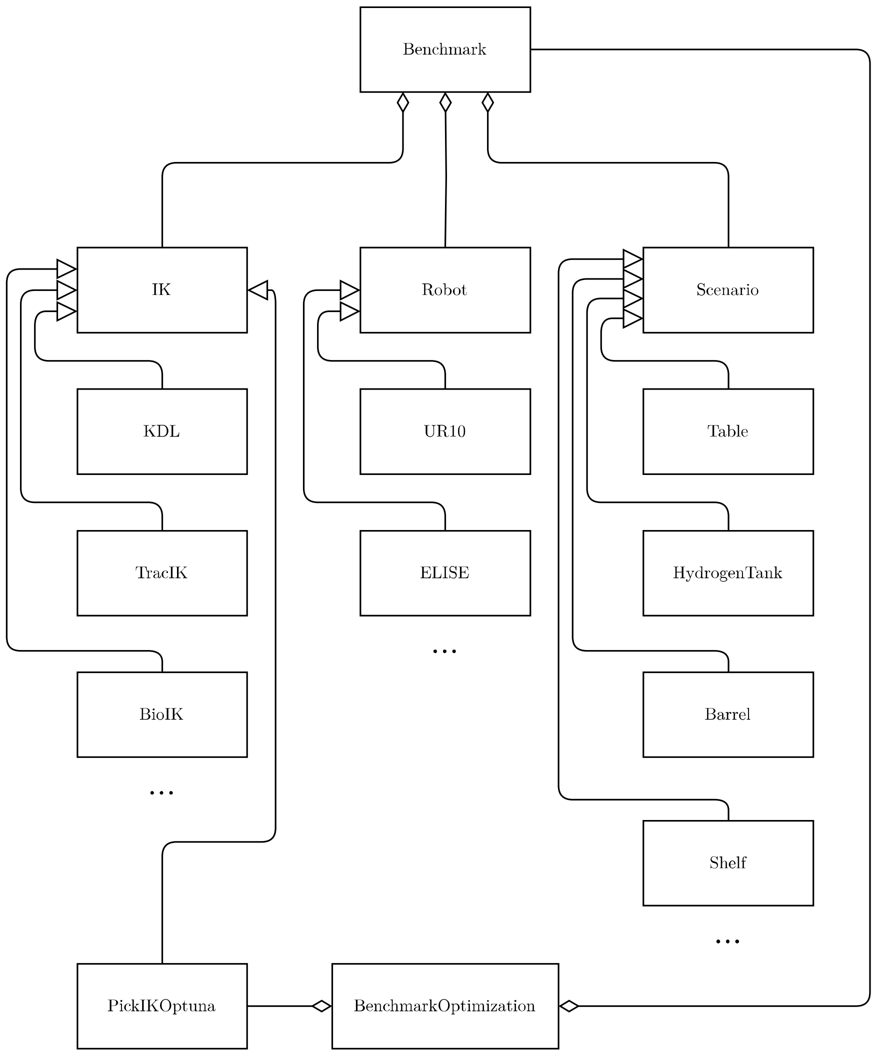


Figure 5.2: UML class diagram of EBIKE

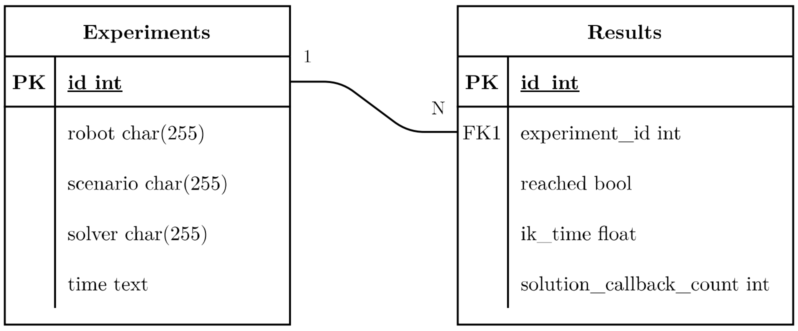


Figure 5.3: Entity-Relationship Model of EBIKE

```
elise
KDL,TracIKDistance,TracIKSpeed,PickIK,BioIK
labels:KDL,TracIK (Distance Mode),TracIK (Speed Mode),PickIK,BioIK
colors:0,1,1,2,3
styles:solid,solid,dotted,solid,solid
barrel
shelf
```

Listing 5.1: Example configuration file for EBIKE's visualization

and robots, using the selected IK solvers. In the REACH configuration, the optimization loop is skipped to only evaluate the single-point inverse kinematics performance. REACHs internal datastructures were modified to capture relevant inverse kinematics statistics (success rate, solve time, and solver iterations described below) and add them to the study results available in Python.

The `IK`, `Robot`, and `Scenario` classes are used to configure the corresponding elements of the study. Each of these classes has subclasses for the different options that can be used. Currently, `IK` classes for KDL, TracIK in the Speed and Distance modes, BioIK in selected modes, and PickIK exist. It is possible to add any other IK solver that is supported by MoveIt. All extensions to current IK solvers that are described in Chapter 8 are implemented as additional IK subclasses. The robots that currently exist are the UR10 and the ELISE robot. The existing scenarios are outlined in Chapter 4.

For optimization, the `BenchmarkOptimization` class is used. It builds on the `Benchmark` class to perform the benchmark and uses another implementation of the `IK` class, `PickIKOptuna`, that makes it possible for Optuna to adapt the parameters used by the solver.

The results of evaluation trials are stored in a results database. The database uses SQLITE and contains two tables. The `experiment` table stores metadata of an experiment, the `results` table stores the evaluation results for every single pose. Figure 5.3 shows the detailed database structure.

Evaluation graphs are generated using PyPlot from the results stored in the database. In order to specify the structure of these plots, configuration text files can be created. For example, the file in Listing 5.1 will generate a graph for the *Barrel* and for the *Shelf* scenario, each containing the data for the five specified solvers on ELISE. Results are averaged over the last three trials, minimum and maximum values are also shown in the graphs. Custom labels, colors, and line styles can be specified.

The source code has been published on GitHub under the open source MIT license[19].

5.3 Example

To give an example for a case where the library gives new insights, benchmarks are created for a UR10 robot arm. First, four inverse kinematics solvers are compared to each other on randomly generated poses. This corresponds to standard IK benchmarks that are often used to evaluate these solvers. Next, the solvers are run on the *SmallTable* scenario. This scenario was chosen because the UR10 arm can reach all target poses. Most of the other scenarios contain points that are too far away or too occluded to be reached by the large tip of the arm. The results of the experiments are displayed in Figure 5.4. The top row shows the success rates when a timeout of 1000ms is used. The bottom row shows the cumulative distribution of the solve rate over time. On the x-axis, the duration it takes for solving a single position is shown. The y-axis indicates which fraction of the target poses could be solved within the duration. As an example, the purple line for PickIK on *Random* shows that about 20% of the targets were solved within 10ms. BioIK and TracIK found no more solutions after approximately 3ms.

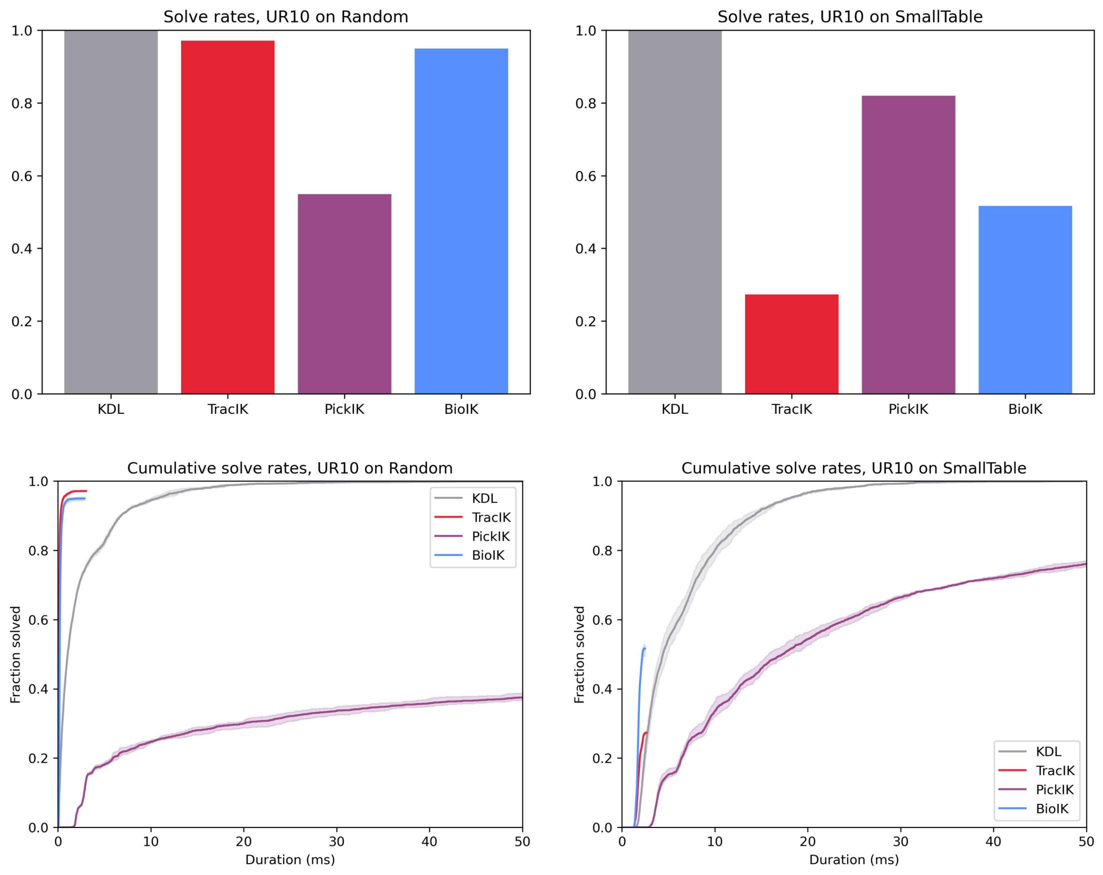


Figure 5.4: EBIKE output for UR10 on *Random* and *SmallTable* scenarios using four different IK Solvers. Note that the bottom two graphs are truncated at 50ms, the full timeout was 1000ms.

[19]https://github.com/DLR-MO/ebike

It is obvious that while the solvers perform well on the random scenario, where they find almost all available solutions, the scenario containing collisions proves much more difficult because the solvers do not properly handle collisions. It also highlights differences between the solvers that were not apparent from the random sample. For example, PickIK performs better on *SmallTable* than on *Random* because it does not handle targets near the workspace boundaries well but restarts the solver when the final solution collides. These differences clearly emphasize the need for software that can reliably compare different IK solvers in surroundings that are not straightforward to solve. The reasons for the different performance of the other solvers and improvements in collision-rich environments are analyzed and developed in the remaining chapters of this thesis.

Chapter 6
Baseline

This chapter presents the baseline performance of state-of-the-art IK solvers on the benchmarking dataset, serving as a reference for the improvements introduced in Chapter 8. The selected IK solvers are those that are usable as MoveIt kinematics plugins. These solvers are KDL, BioIK, PickIK, and TracIK. For KDL, only one variant of the solver exists. As described in Section 3.1.2, only the "Speed" mode of TracIK is used, returning the first solution that was found. PickIK supports two modes, "local" and "global", where the former is a gradient descent implementation, the latter an evolutionary algorithm. For the evaluations, only the "global" mode is used. The most modes are available for BioIK. They are described in Section 3.1.3, but only the default mode is used in these evaluations. Other BioIK variants are evaluated in Section 7.3.

First, as a general comparison, the solvers are compared in a collision-free environment. For this, the *Random* scenario is used, where 1000 valid joint states are randomly sampled to generate reachable end effector poses. These poses are then used as input to the IK solvers and their success rates and solve times are recorded.

Then, the real-world scenarios of the benchmarking dataset are evaluated. The goal is to find a configuration where the end effector reaches the target pose on the surface of the object, touching it, but not colliding with it in any place. The objects are described in Chapter 4. All evaluations are performed on the ELISE robot mounted on an UR10 arm.

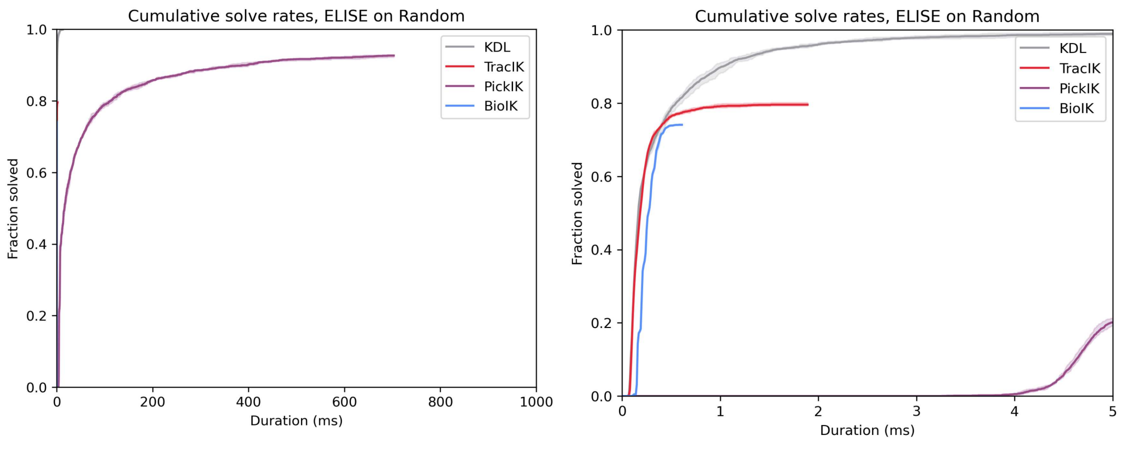


Figure 6.1: Cumulative solve rates for the *Random* scenario, complete timeout and detailed view of the first 5ms

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	98.9%	99.9%	100.0%	100.0%	0.475ms
TracIK	79.6%	79.6%	79.6%	79.6%	0.205ms
PickIK	20.1%	42.6%	79.0%	92.7%	55.250ms
BioIK	74.1%	74.1%	74.1%	74.1%	0.252ms

Iterations	1	2	3	5	10	100	Mean
KDL	77.2%	93.2%	97.6%	99.4%	100.0%	100.0%	1.343 its
TracIK	79.6%	79.6%	79.6%	79.6%	79.6%	79.6%	1.000 its
PickIK	78.4%	86.8%	89.0%	90.4%	91.7%	92.7%	1.422 its
BioIK	74.1%	74.1%	74.1%	74.1%	74.1%	74.1%	1.000 its

Table 6.1: Solve rates after given durations and solver iterations on *Random*

From the cumulative solve rates on *Random* presented in Figure 6.1, it is apparent that three out of four solvers find solutions very quickly. KDL, TracIK, and BioIK manage to find about 80% of the solutions after less than 1ms. KDL is the only algorithm that manages to find all solutions. TracIK and BioIK stop to find new solutions relatively quick, PickIK continues to a maximum duration of about 700ms, but also does not manage to find all solutions.

The tabular overview in Table 6.1 shows the solve rate within a range of durations (5ms, 10ms, 100ms, 1000ms) and the mean solve time for all successful solves. This format was used to quantify how quickly the solvers find solutions. The mean value should only be considered in combination with the number of solutions found after the full timeout because a solver that quickly stops finding solutions but does not find all of them has a better mean solve rate than a solver finding all solutions but more slowly.

The second table shows the number of iterations done by the algorithms. One iteration here means that one call to the solution callback, usually performing collision checking, was done. In the case of the *Random* scenario, the solution callback was only used to detect self collisions. From the results table, it can be seen that TracIK and BioIK only use a single iteration and fail immediately if the solution callback fails. The results also show that while PickIK is much slower than KDL, the algorithm performs better in the first iteration. There, 78.4% of the solutions are found, compared to 77.2% found by KDL in the first iteration. KDL then quickly proceeds to increase the fraction of solutions found in the next iterations and surpasses PickIK. The measure of solver iterations therefore helps to evaluate the difference between implementation and theoretical solver performance. For example, a solver that considers collisions might take more time but work better on fewer solver iterations, showing that there is a high potential that could be unlocked if the implementation is improved.

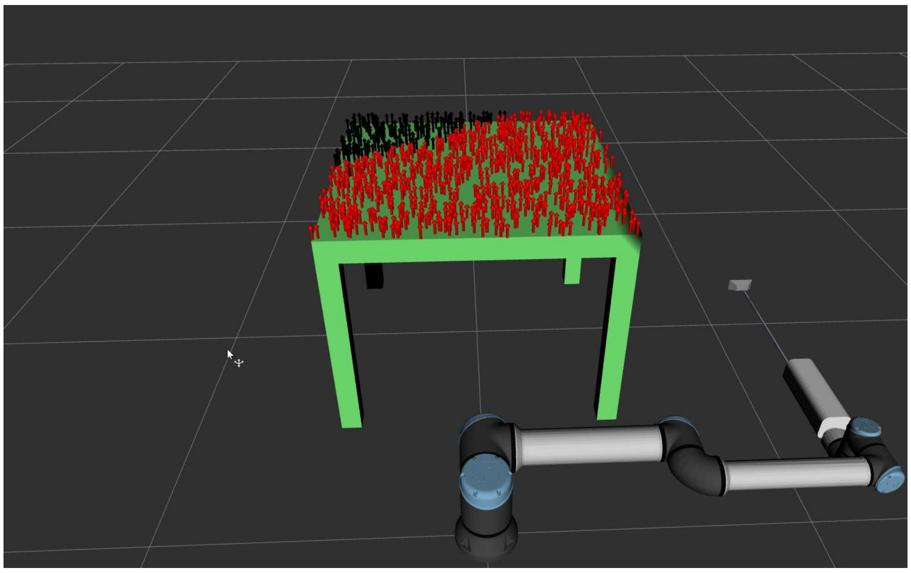


Figure 6.2: Partial coverage (80.4%) of *Table* using TracIK in rViz

For the collision scenarios, similar behavior with worse performance can be seen. Figure 6.2 shows the results of TracIK on *Table*. The table was not fully covered because collision-free configurations could only be found for the front part of the table, not for the back part. Similar behavior was observed for BioIK. Figure 6.3 shows the quantitative results for *Table*. Again, TracIK and BioIK did not find all results and PickIK did not find any solutions that took longer than 500ms. In this scenario, the time onset is different than in the random scenario, no solver found solutions in the first 1.5ms, whereas in the random scenario, most solutions were already found within that time frame. The results for *SmallTable* and *TableObjects* look similar to those of *Table*. Detail views for *Barrel* and *Shelf* are shown in Figure 6.4. In these scenarios, the solvers find solutions very slowly because of the high number of colliding objects. Only KDL manages to find all solutions within the timeout of 1000ms. The hydrogen tank scenarios are even more extreme, KDL is the only solver that manages to find a reasonable fraction of solutions, and even these are at 56.8% and 15.2% for the *HydrogenTank* and *HydrogenTankSmall* scenarios, respectively. The plots for all scenarios can be found in Appendix A1, the quantitative results in Appendix A3.

The results of the solvers show that there is a clear need for improvement of IK solvers in collision-rich environments. Even relatively simple scenarios pose problems for state-of-the-art solvers. The hydrogen tank scenarios are not solvable by them within a reasonable amount of time.

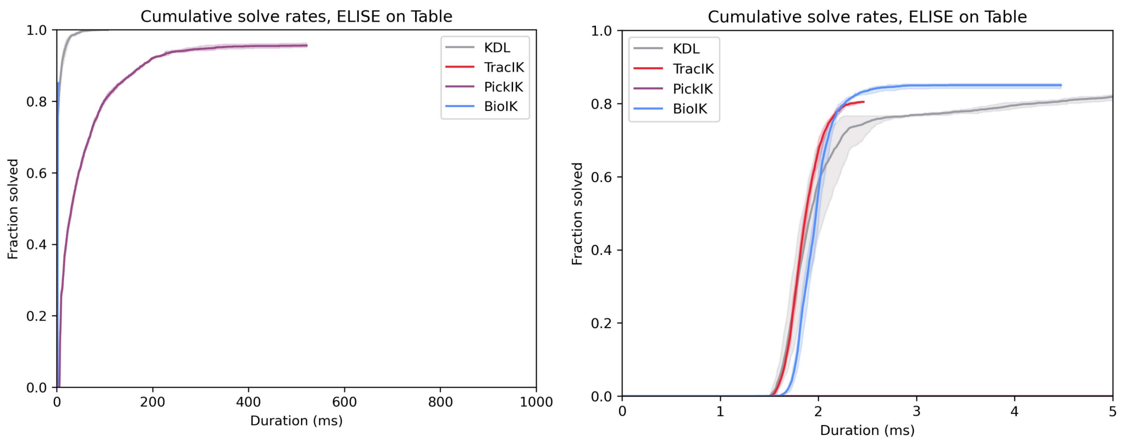


Figure 6.3: Cumulative solve rates for the *Table* scenario, complete timeout and detail until 5ms

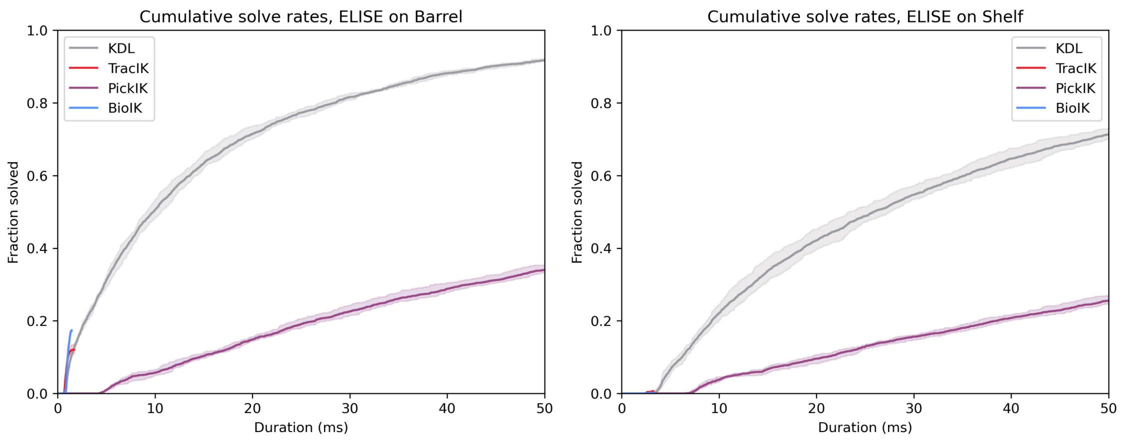


Figure 6.4: Solver performance on *Barrel* and *Shelf* scenarios, detail view until 50ms. On *Shelf*, TracIK and BioIK find almost no solutions, the graphs are therefore barely visible.

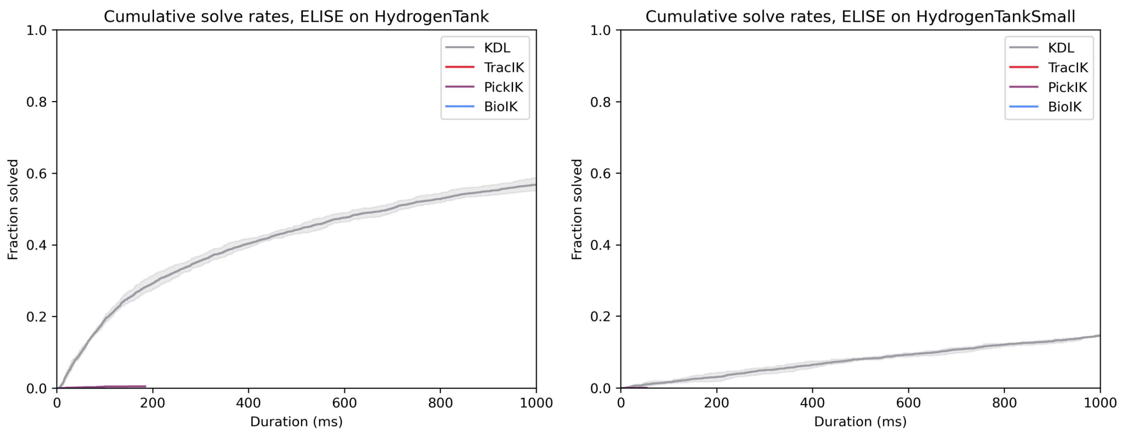


Figure 6.5: Hydrogen tank scenarios. Only KDL can solve a significant number of targets.

Chapter 7

Implementation

This chapter describes the implementation of the MoveIt plugins used in the evaluations in Chapter 8, focusing on the two inverse kinematics solvers RelaxedIK and BioIK. RelaxedIK is an optimization-based solver, with its C++ implementation developed as part of this work. BioIK 2, a memetic solver that combines evolutionary algorithms and optimization, was already available as a MoveIt plugin but was modified to make previously hard-coded parameters configurable.

The code developed in this thesis is implemented for the ROS 2 framework and the MoveIt motion planning library. The inverse kinematics solvers are created as MoveIt kinematics plugins.

7.1 MoveIt Kinematics Plugins

Kinematics plugins for MoveIt are C++ classes derived from the `KinematicsBase` class. This class defines several methods that should be implemented. The `initialize` method can be used to set up the solver and initialize data structures for later use. As a kinematics plugin is instantiated per joint model group, pre-calculations regarding this group can already take place. The main methods of the class are the `getPositionIK` and `searchPositionIK` methods. `getPositionIK` should be used for local solvers only, i. e., solvers that search for a solution in proximity to the previous solutions. Random restarts and jumps in joint space are not allowed as this method is supposed to be used for trajectory tracking. The `searchPositionIK`, by contrast, is allowed to find any solution to the IK problem it receives. In MoveIt's code base, mostly `searchPositionIK` is used. Most notably, it is the function that is used in the `setFromIK` method of the RobotState object, which is the only use of inverse kinematics that is exposed by MoveIt.

The work in this thesis only uses the `searchPositionIK` method as it is the more general one. One signature of the method is shown in Listing 7.1. There are multiple other, overloaded uses of the function, but they do not contain more information required for this thesis.

```
bool
searchPositionIK(const geometry_msgs::msg::Pose& ik_pose,
                 const std::vector<double>& ik_seed_state,
                 double timeout,
                 std::vector<double>& solution,
                 const IKCallbackFn& solution_callback,
                 moveit_msgs::msg::MoveItErrorCodes& error_code,
                 const kinematics::KinematicsQueryOptions& options) const;
```

Listing 7.1: Signature of `searchPositionIK`[20]

The `ik_pose` parameter is the target for the IK solver. The `ik_seed_state` is the initial seed used by the solver. It is initialized with the current state of the robot. The `timeout` is the maximum time available, in seconds. `solution` is the output parameter of the function. `solution_callback` is a function provided to the solver that should test whether a found solution is valid. It is mostly used for collision checking. `error_code` is an output parameter giving further information on a failed execution. `options` are more options given to the solver. They can, for example, be used to specify that a problem only has to be solved approximately.

MoveIt kinematics plugins are not collision-aware and have no information about the environment. Usually, the solution callback is used for collision checking, but it only provides a binary valid/invalid decision and no further information. Custom goals can be used to add this information by using the `options` parameter, but this requires a non-standard extension of the `KinematicsQueryOptions` struct. In Tesseract, inverse kinematics plugins are also not environment-aware “to simplify the process of implementing both forward inverse kinematics”[21].

7.2 RelaxedIK C++

The C++ implementation of RelaxedIK is done within the ROS 2 framework. The software itself is designed as a plugin for the MoveIt motion planning library and called relaxed_ik_cpp.

In relaxed_ik_cpp, the `searchPositionIK` function is defined and creates an instance of an nlopt-cpp optimization problem that is then solved using SLSQP. The problem that is minimized consists of multiple objectives that are managed by an objective master. All objectives provide their own cost function and are assigned a weight. The weighted sum of the individual costs result in the total cost. In addition, a gradient is calculated by the objective master. The gradient is computed numerically by changing all parameters by a small value (10^{-7}) and recalculating the cost functions. The loss function differences divided by epsilon build the gradient.

[20]https://github.com/moveit/moveit2/blob/2.10.0/moveit_core/kinematics_base/include/moveit/kinematics_base/kinematics_base.h#L253-L257

[21]https://tesseract-docs.readthedocs.io/en/latest/_source/core/overview/index.html#tesseract-kinematics, accessed 09/09/2024

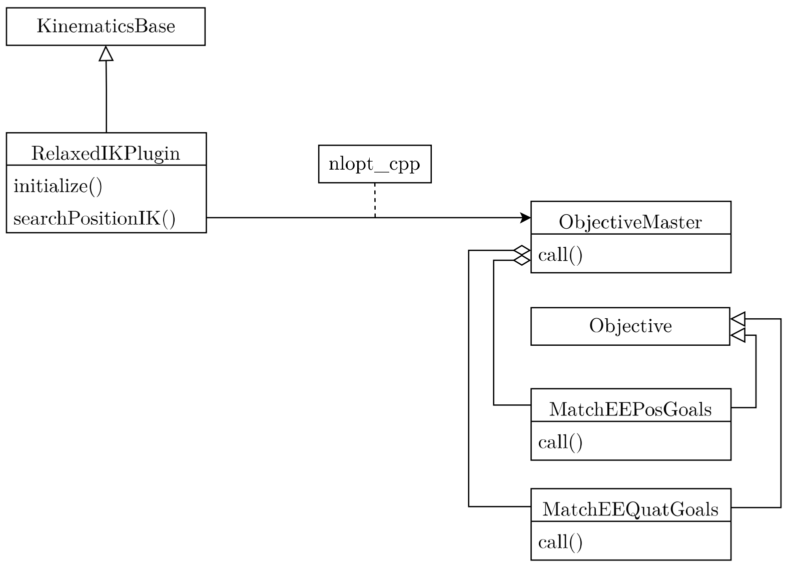


Figure 7.1: Class diagram of RelaxedIK. `RelaxedIKPlugin` inherits from MoveIt's `KinematicsBase`. It uses `nlopt_cpp` to call `ObjectiveMaster` which holds multiple `Objectives`, calculates their function values and approximates a gradient.

In contrast to BioIK, an advantage of RelaxedIK is that the objective functions are called much less often. An analysis on the *Random* scenario, which both solvers can solve very quickly, shows the difference: BioIK requires an average of 20300 objective function calls on the 1000 targets of *Random*, RelaxedIK only requires an average of 582 calls. Naturally, the number of objective function calls depends on the difficulty of the problem and whether the solve was successful or failed within the given timeout.

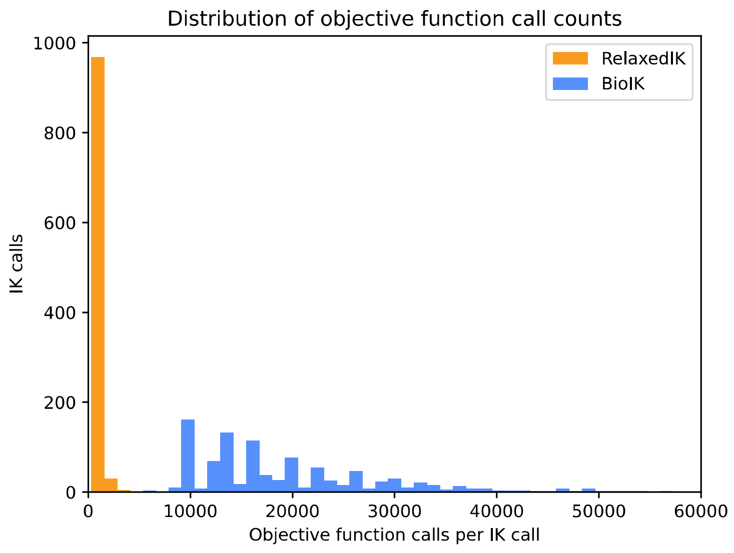


Figure 7.2: Histogram of objective function calls for BioIK and RelaxedIK. The plot shows how many of the 1000 targets of the *Random* scenario required how many objective function calls. RelaxedIK consistently requires fewer calls.

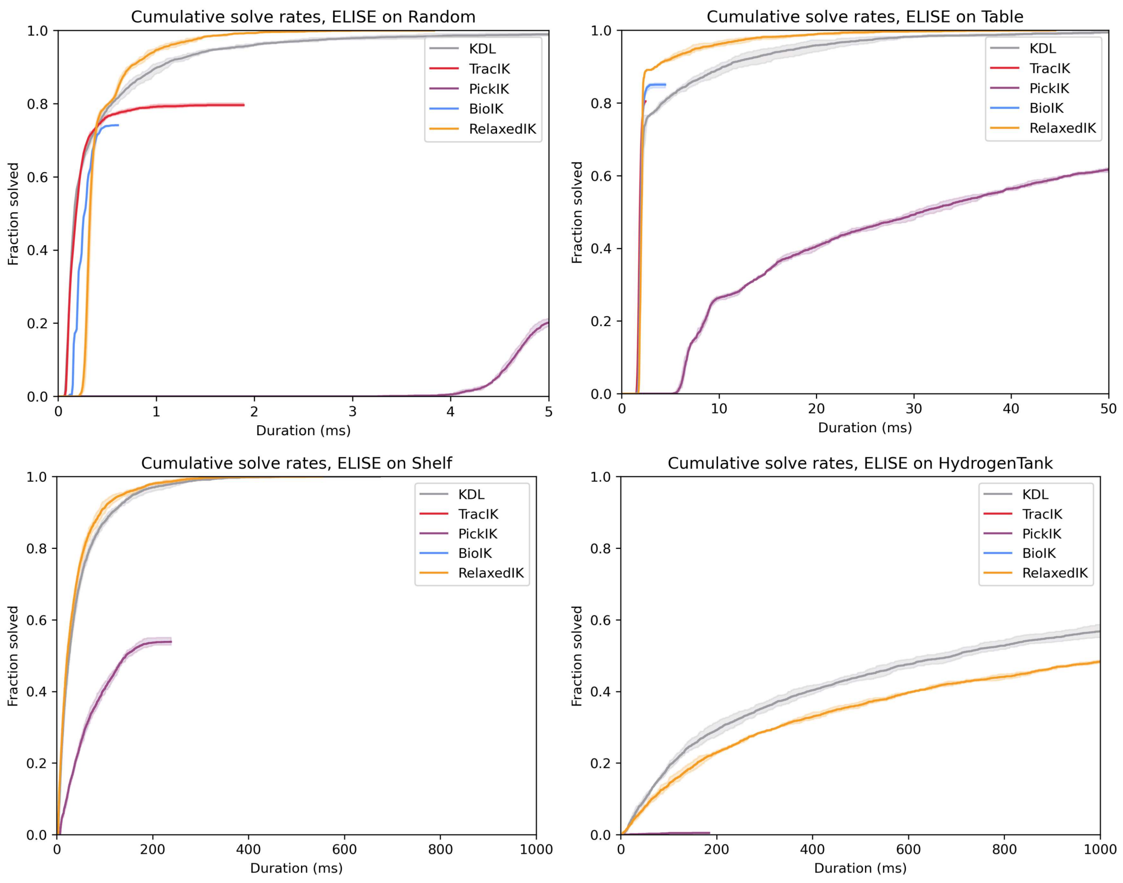


Figure 7.3: Comparison of RelaxedIK to other MoveIt IK solverson *Random*, *Table*, *Shelf*, and *HydrogenTank*. RelaxedIK is the fastest solver to find all solutions on simpler scenarios and is only outperformed by KDL on more complex scenarios. Appendix A2 shows the results on more scenarios.

In the implementation of RelaxedIK, the same result verification method as for TracIK and BioIK is used, i. e., the position and axis-angle difference vectors are not larger than 10^{-5} in any dimension. No verification method for other goals has been implemented.

Figure 7.3 shows RelaxedIK on selected scenarios in comparison to the current IK solvers available for MoveIt. RelaxedIK outperforms all existing solvers on the *Random*, *Barrel*, *Shelf*, and *Table* scenarios. On *TableObjects* and *SmallTable*, its performance is similar to KDL, on *HydrogenTank* and *HydrogenTankSmall*, KDL performs better. In contrast to KDL, RelaxedIK's goals can be adapted and tuned for specific problems. Combined with its high performance, this makes it a strong candidate for further improvements in Chapter 8.

7.3 BioIK

BioIK is also a MoveIt Kinematics Plugin and therefore shares some general structures with RelaxedIK. BioIK has been extensively profiled and optimized in [Rup+18]. For example, the default MoveIt RobotState class is replaced by custom forward kinematics as forward kinematics are the part of the calculation that took the most time. The BioIK implementation has been adapted to make some parameters configurable, making it pos-

Parameter	**Description**	**Default Value**
`population_size`	Number of possible parents	2
`child_count`	Number of children	16
`elite_count`	Number of elites	1
`threads`	Number of parallel (independent) threads	4
`species_count`	Number of populations per thread	2
`memetic_evolution_gens`	Number of iterations for evolution	8
`memetic_opt_gens`	Number of iterations for optimization	8

Table 7.1: Configurable parameters for BioIK

sible to perform automatic parameter optimization using EBIKE. These parameters are listed and described in Table 7.1.

Allowing changes to the population size raises the question for how the parents should be selected if more or less than two individuals are available. It is clear that the behavior should not be changed when the population size is set to 2, where the better individual is used as parent1 and the worse individual is used as parent2. The decision is therefore made to always select two distinct parents from the population and make the better one parent1. The selection happens randomly and individually for each child to be created.

In addition to the parameters, BioIK's optimization solvers have been made available for ROS 2.

Chapter 8

IK Solver Improvements

This chapter describes the improvements made to existing inverse kinematics (IK) solvers. Most changes are applied to RelaxedIK and BioIK because these solvers accept cost functions. Although PickIK also supports cost functions, it uses the same principle as BioIK but with significantly lower performance.

The evaluation is done using the EBIKE library developed in Chapter 5. An IK solver is evaluated by consecutively attempting to solve 1000 different, randomly generated, points on the surface of the objects. For the *Random* scenario, 1000 target positions that are randomly distributed across the joint space are given to the solver. All target poses are solved independently of each other. No seed is given to the solvers and the solver timeout is set to one second, unless explicitly stated otherwise. All results are averaged across three trials, with the evaluation graphs displaying the average, minimum, and maximum values. The solvers used for evaluation are explained in detail in Section 3.1, their termination criteria are also explained there. For TracIK, BioIK and RelaxedIK, the linear error is always below $1.73 \cdot 10^{-5}$ meters, the angular error is below $1.73 \cdot 10^{-5}$ rad $\approx 10^{-3}$ degrees. For KDL and PickIK, the maximum linear or angular error is 10^{-5}. This difference is due to the implementation of the error function but it is not deemed critical for the evaluations. In the context of this evaluation, BioIK always refers to the default (bio2_memetic) mode of the C++ reimplementation of BioIK (see Section 3.1.4).

All evaluations are performed on an AMD Ryzen 9 7950X 16 core / 32 threads processor running Ubuntu 22.04 and ROS 2 Rolling.

8.1 Rejection Sampling

None of the current solvers except KDL and RelaxedIK handle failed solution callbacks correctly. If the solution callback failed, these solvers return immediately with a negative result even when the timeout is not exceeded. This is critical for collision avoidance. All of the solvers do not internally handle collisions, therefore the solution callback is the only way in which they are treated. Failing the IK when the callback fails leads to missing solutions. To circumvent this problem, KDL and RelaxedIK perform rejection sampling: When an IK solution is found but the solution callback fails, the solver is restarted with a random seed. This is repeated until a solution passing the callback is found or the timeout is exceeded. TracIK and BioIK 2 do not perform rejection sampling; PickIK does, but the implementation is wrong[22].

[22]https://github.com/PickNikRobotics/pick_ik/pull/73

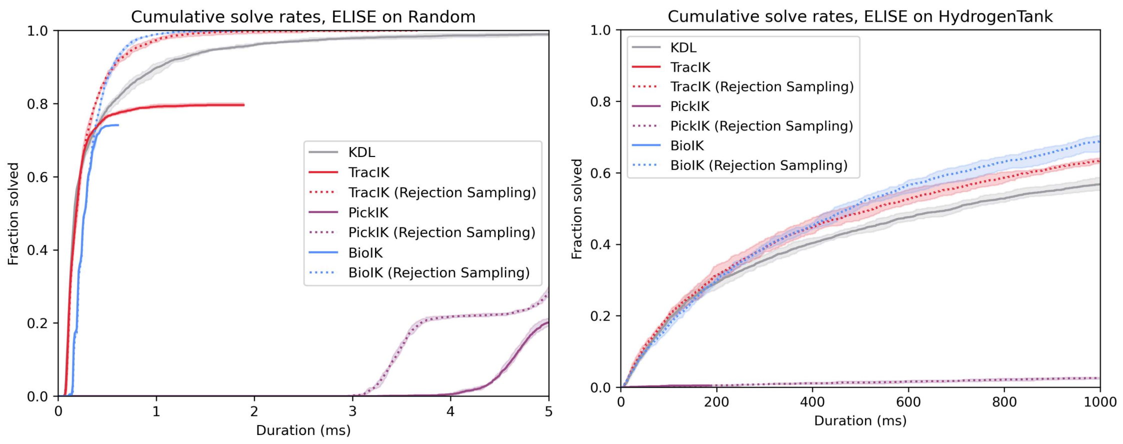


Figure 8.1: Results of IK solvers with and without rejection sampling. KDL is added as a baseline. On *HydrogenTank*, TracIK and BioIK do not solve a single target.

Adding rejection sampling to existing solvers is relatively easy. The complete code of the solver except for setup is wrapped in a while loop that runs as long as time is still available. Whenever the code previously returned without a solution, the robot is reset to a random configuration and the solver is restarted with an updated timeout.

Figure 8.1 shows the improvements made by adding rejection sampling. On the left, the results on *Random* are shown. The difference for TracIK and BioIK is clear, they previously stopped optimizing at less than 80%, with the change they continue until all solutions are found, both surpassing KDL on their way. While TracIK initially finds solutions faster, BioIK is the first solver to find all solutions. PickIK does not show a difference for small durations, but where the solver previously stopped finding new solutions after approximately 150ms, the correct implementation of rejection sampling continues finding solutions.

Table 8.1 confirms the results. All solvers with rejection sampling correctly implemented find all solutions when the full timeout is available, only PickIK still does not find all solutions.

Duration	Rejection Sampling?	5ms	10ms	100ms	1000ms
KDL	✓	98.9%	99.9%	100.0%	100.0%
TracIK	×	79.6%	79.6%	79.6%	79.6%
	✓	100.0%	100.0%	100.0%	100.0%
PickIK	×	20.1%	42.6%	79.0%	92.7%
	✓	28.0%	45.9%	82.5%	95.8%
BioIK	×	74.1%	74.1%	74.1%	74.1%
	✓	100.0%	100.0%	100.0%	100.0%

Table 8.1: Results of IK solvers with and without rejection sampling on *Random*

Solver	Mean Iterations without Rejection Sampling	Mean Iterations with Rejection Sampling
KDL	-	1.343
TracIK	1.000	1.315
PickIK	1.422	2.356
BioIK	1.000	1.360

Table 8.2: Mean solver iterations with and without rejection sampling on *Random*

Regarding solver iterations, we can see in Table 8.2 that TracIK now uses 1.315 iterations on average, BioIK uses 1.36 iterations on average and PickIK increased its average iterations from 1.422 to 2.356.

The right panel of Figure 8.1 shows that TracIK and BioIK with rejection sampling manage to outperform KDL. All other scenarios show similar improvements; TracIK and BioIK with rejection sampling consistently show similar but slightly better solve rates than KDL.

These evaluations show that rejection sampling is a very simple strategy to increase the solve rates for problems that use a solution callback to handle self- or environment collisions.

All following situations where the solvers BioIK, TracIK, or PickIK are mentioned use the implementations with rejection sampling. RelaxedIK and KDL already use rejection sampling in their base implementation.

8.2 Collision Distance

As a first approach to incorporate collision checking directly into the IK solver, the distance to collision was used. This function is already implemented in MoveIt as the `distanceToCollision` function. Similar to the approach in CollisionIK [Rak+21], the function $\left(\frac{2\varepsilon}{\varepsilon+d}\right)^2$ is used to weigh the distance d. ε serves as a cut-off point between collision and non-collision; if d is equal to ε, the cost is exactly 1. The output of this cost function was used as input for the groove loss and the same parameters as in CollisionIK have been used ($s = 0, c = 2.5, n = 1, r = 0.035, g = 4$).

In CollisionIK, the distance from all links to all obstacles is used. While this potentially results in a higher number of distance calculations, the results might be more usable by the algorithm because a clearer gradient might be present if a change to the configuration only brings parts of the robots further away from collision. However, the implementation of this approach in MoveIt with FCL proved difficult. Therefore, only the overall distance to a collision was used. This approach can help to steer away of collisions, but all colliding states have the same cost and states where the end effector is touching or almost touching the surface of an object have almost the same cost as a collision.

To reduce the complexity of collision checks, MoveIt's allowed collision matrix (ACM) can be leveraged. This matrix specifies which collisions are checked and which are ignored. In this case, the UR arm can be added to the matrix so that all collisions of the arm are

Solver	Successful	Total time	Calls to objective function	Time spent on collision checking	Time per collision check
RelaxedIK	✓	334.4ms	63100	×	×
	×	1001.2ms	206000	×	×
RelaxedIK with CollisionDistance	×	1011.3ms	547	1009.1ms (99.8%)	1.866ms
RelaxedIK with ACMCollisionDistance	✓	463.2ms	820	457.6ms (98.7%)	0.583ms
	×	1002.8ms	2280	991.2ms (98.8%)	0.448ms

Table 8.3: Time spent on collision checking in RelaxedIK (averaged over the 1000 IK queries of *HydrogenTank*). Results are grouped by their query success and rounded.

ignored. This should speed up distance calculations because only collision distance from ELISE's three links to the mesh are calculated.

Implementing and executing this idea shows that this approach is not feasible. Neither BioIK nor RelaxedIK manages to find a single solution within the one-second timeout. Using the ACM to reduce the number of possible contacts, RelaxedIK is able to find around 0.2% of the solutions within the time frame.

In order to better understand the reasons why so few solutions are found, these results are investigated for RelaxedIK. Table 8.3 shows quantitative results of RelaxedIK on *HydrogenTank*. When RelaxedIK is run with the default PoseGoal, the objective function is called very often. When a solution is found, it is called on average 60k times, when no solution is found and the full timeout is used, it is called 200k times. This corresponds to approximately 200 objective function calls per millisecond for both successful and unsuccessful queries. When collision distance checking is used, the number of objective function calls is greatly reduced to around 500 calls. The reason for this is that a single collision distance check requires almost 2ms, so that 99.8% of the solver time is spent within the collision distance checking call. When the ACM is used to disable most collisions, the distance query time is reduced by almost 70%. Therefore, more calls to the objective function can be made and the solver is more likely to find results. In the experiment shown in the Table 8.3, this was successful for 0.2% of the targets. Without the collision distance queries, some solutions are also found within 500-800 objective function calls, therefore it makes sense that these are also found using the CollisionDistance goal.

For BioIK, the number of objective function calls is even more extreme: When a result was found, the average was approximately 1.9 million evaluations, when no solution was found, the objective function was evaluated 6.6 million times. Even on the random scenario on which the solver is optimized, the average number of calls was 20k and the minimum number of calls required to find any solution was approximately 6000. This shows that even with a perfectly shaped objective function that requires as much time as the collision distance check, no solution would be found.

The experiments showed that using the collision distance, even when the number of collision checks is greatly reduced, is too cost-expensive because a single collision distance query already requires around 0.5ms, limiting the number of objective function calls.

8.3 Penetration Depth

Instead of using the distance to a collision, the penetration depth can be used as a measure for collision cost. While this does not help the solver steer clear of collisions, it helps in guiding it away from collisions. In practice, this might be more relevant because it makes it possible for the robot to touch or hover over a surface without inflicting high cost. The function $\left(\frac{d}{\varepsilon}\right)^2$ is used, resulting in a quadratic cost for higher penetration depths and using an epsilon guiding the cost function such that a penetration depth of ε yields a cost of 1. For RelaxedIK, the groove loss is used, with the parameters $s = 0, c = 0.01, n = 1, r = 10, g = 2$. These parameters are the same as for the pose goals, except that c was adapted to punish relatively small penetration values more strongly.

Figure 8.2 shows the results of using penetration depth with BioIK and RelaxedIK. Clearly, both solvers are slower when penetration depth is used than when no penetration depth is used. However, when looking at solver iterations, the solvers using penetration depth require fewer iterations to find solutions than the solvers without penetration depth. This shows that using penetration depth is a good metric for a cost function because it does help the solver find solutions.

The profiling results for RelaxedIK can be seen in Table 8.4. Penetration depth calculation is much faster than collision distance queries, taking around 11ns compared to 1.9ms. Therefore, the solver can reach more solver iterations and is able to find more solutions. Still, more than 90% of the solver time is spent within the penetration depth query, making it more difficult to explore and exploit as well as when no penetration depth query is used. Once again, this effect is more extreme for BioIK where more calls to the objective

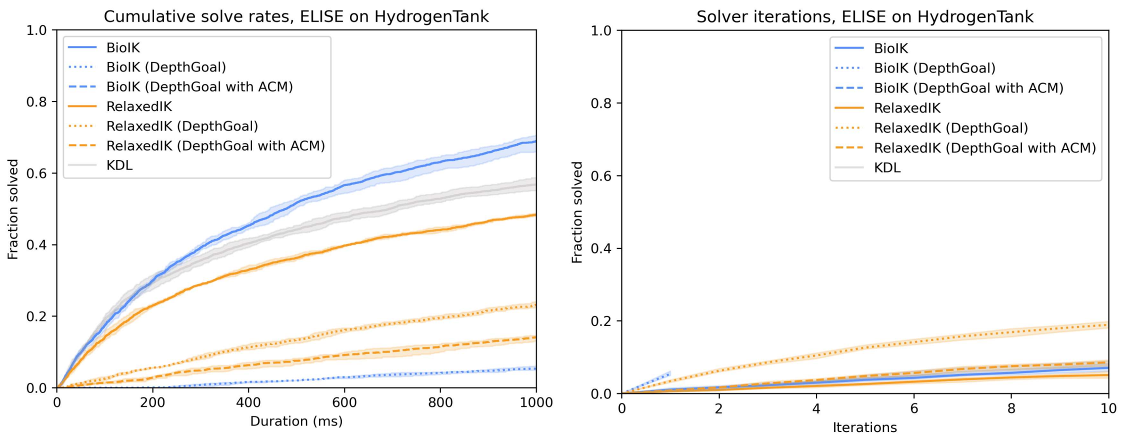


Figure 8.2: Penetration depth results on *HydrogenTank*. While the solvers using penetration depth perform worse in terms of time, they perform better in terms of calls to the solution callback. BioIK (DepthGoal with ACM) did not solve a single target.

Solver	Successful	Total time	Calls to objective function	Time spent on penetration depth calculation	Time per penetration depth calculation
RelaxedIK	✓	334.4ms	63100	×	×
	×	1001.2ms	206000	×	×
RelaxedIK with Penetration Depth	✓	440.7ms	36100	393.4ms (88.2%)	0.011ms
	×	1000.1ms	84900	903.6ms (90.3%)	0.011ms
RelaxedIK with Penetration Depth and ACM	✓	468.4ms	79900	346.9ms (72.9%)	0.004ms
	×	1000.1ms	176000	747.1ms (74.7%)	0.004ms

Table 8.4: Time spent on penetration depth checking in RelaxedIK (averaged over the 1000 IK queries of *HydrogenTank*). Results are grouped by their query success and rounded.

function are necessary. While the solver iterations for both are similar, indicating that using the penetration depth metric works equally well for both solvers, BioIK is slowed down even more by the queries.

As in the previous section, the penetration depth query has also been facilitated by employing the ACM. While the speed of the penetration depth checking is reduced, the solver also finds fewer solutions (14.1% instead of 23.1% with a timeout of 1000ms). This shows that the penetration depth when only the endoscopic part is considered does not fully capture the extent of the problem.

In order to better evaluate the potential of the penetration depth solver and the collision distance solver, they are run again with a larger timeout of 10s. The results are shown

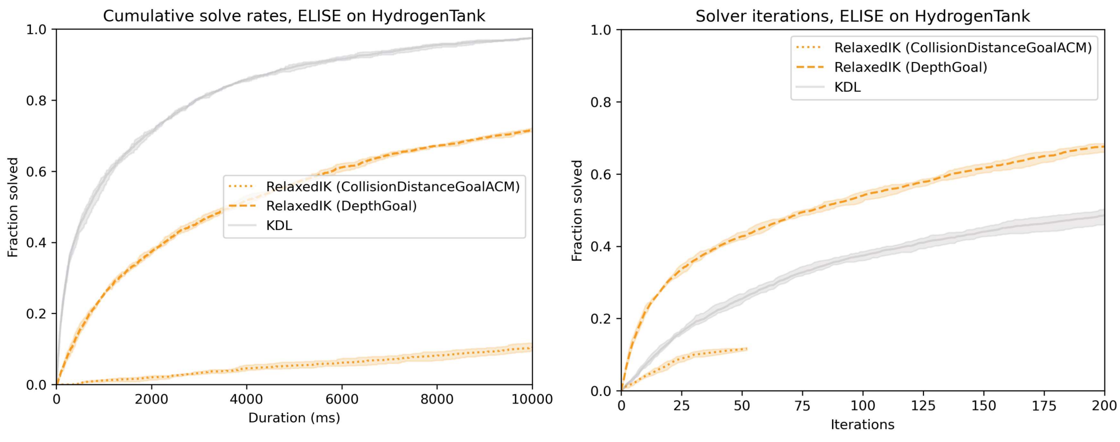


Figure 8.3: Penetration depth solver and collision distance solver with a timeout of 10s

in Figure 8.3. In terms of time, both solvers remain outperformed by KDL. Regarding solver iterations, however, the penetration depth solver remains in a constant lead to KDL, while the collision distance solver is not able to compete. This emphasizes the result that penetration depth is a better cost function than collision distance, regardless of their computation times.

8.4 Improving Distance Query Efficiency

Unfortunately, distance and penetration depth queries in FCL are quite expensive compared to simple collision checking. As shown in the previous sections, distance queries are especially expensive. Tests of two simple colliding, non-convex meshes gives approximate times of 3µs per collision checking call compared to 60µs per distance call. Penetration depth calculation, in contrast, does not require a significant time overhead compared to collision checking. These results show that it is difficult to perform a high number of distance queries in the short time frame available for an inverse kinematics solver. An approach that could be used to improve distance checking time would be padding-based binary search. FCL makes it possible to give objects padding, i. e., a layer around the object where a penetration already counts as a collision. Normally, padding is used to ensure safety margins around objects and avoid them moving extremely close to each other. Instead, padding can also be used to perform binary search of an objects collision distance. In the time required to perform a single distance call, 20 collision checking calls can be performed, each cutting the current distance accuracy in half. Starting with an accuracy of for example 40 centimeters, as objects that are further away from each other than that can be considered irrelevant, 20 bisections would result in an accuracy of 380nm which is much more precise than required. Instead, ten queries would already bring the accuracy to a sub-millimeter resolution, cutting the distance query time in half. Unfortunately, this approach is not possible because padding is not correctly implemented in FCL. A patch has been suggested but is not available for ROS 2 at the time of writing[23]. Even when the patch is available, the gradient created with this method might not be usable because only discrete steps of possible distances are possible. In addition, the faster collision distance queries might not yield better overall results because even with larger timeouts tried in the previous section, the collision distance did not serve as a good cost function.

Since a lot of the calculations in RelaxedIK stem from the calculation of the gradient, it would also greatly improve the query speed if these calculations could be reduced. When distance or penetration depth queries are performed with FCL, the contact normal is also provided, i. e., the vector that points from one colliding object to the other. This vector could be used to calculate an approximation of the gradient because it gives the direction in which the penetration depth or collision distance decreases or increases for each collision object. Such an improvement could increase the efficiency of the solver, but as the solver quality measure of solver iterations instead of solve time is already independent of the solver efficiency, this approach was not implemented within the scope of this thesis.

[23]https://github.com/moveit/geometric_shapes/pull/238

8.5 RCM Objective

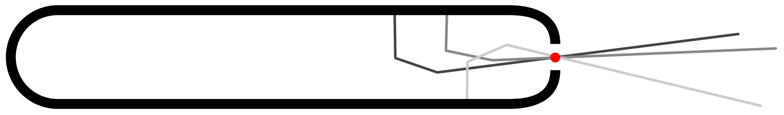


Figure 8.4: Illustration of different targets solved by traversing a remote center of motion (in red)

A remote center of motion can be added as an objective as well. For this, the remote center of motion has to be defined manually for each object in the dataset. When using the equation for p_{rcm} from Section 2.12, the link that has to pass through the remote center of motion has to be defined in advance. If it is not desired to constrain the robot to a single link passing through the RCM, the formulation has to be adapted to work for all links. Therefore, a clamp operator is included into the function to clamp the value of $p_r \hat{p}_s$ to a value between zero and one. This ensures that the projection of the RCM always lies on the current link and not anywhere else on the corresponding line. The complete objective function is then taken as the minimum of all the distances between the target point and the projected RCM point.

$$f_{\text{RCM}} = \min_{l_1, l_2 \text{ consecutive}} \|p_{\text{trocar}} - p_{\text{rcm}}(l_1, l_2)\|$$

with

$$\begin{aligned} p_{\text{rcm}}(l_1, l_2) &= p_{l_1} + \operatorname{clamp}\Big(p_{l_1} \hat{p}_s(l_1, l_2), 0, 1\Big) \hat{p}_s(l_1, l_2) \\ \hat{p}_s(l_1, l_2) &= \frac{p_{l_2} - p_{l_1}}{\|p_{l_2} - p_{l_1}\|} \end{aligned}$$

A disadvantage of this method is that it assumes all links to be straight. It is therefore not usable for robots with curved links such as Eeloscope2 (see Figure 2.9). Additionally, it would be advantageous to remove manual annotation requirements from the process. However, automatically detecting the orifice in a mesh is not trivial and out of scope for this thesis.

For RelaxedIK, the output of the objective function is wrapped in the groove loss function with parameters $s = 0, c = 0.01, n = 1, r = 10, g = 2$. The function output is used directly

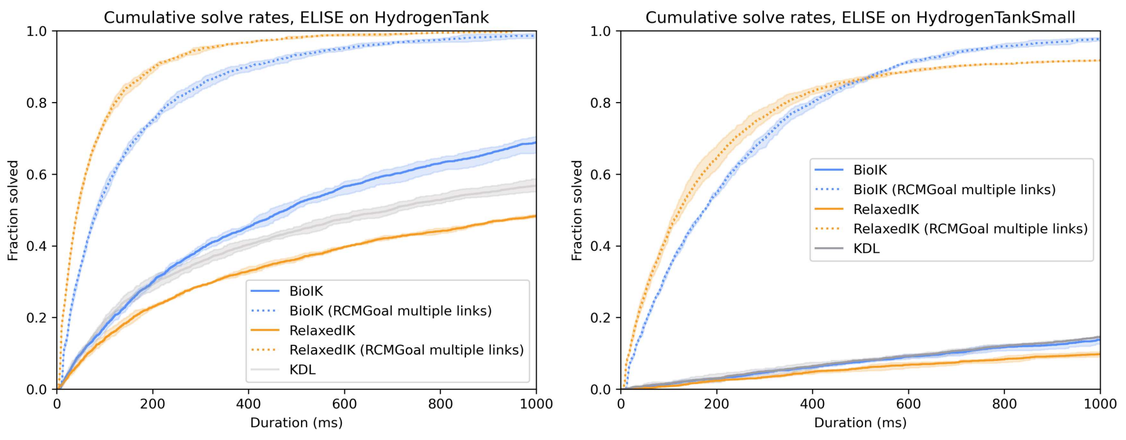
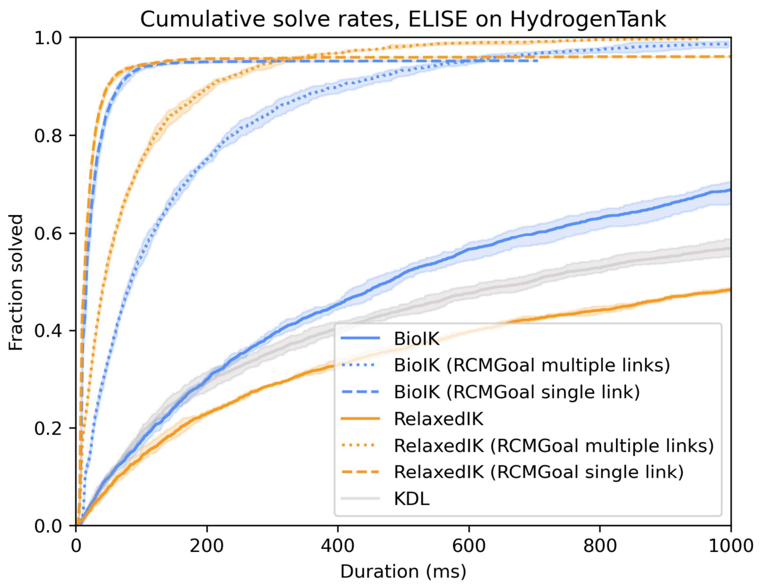


Figure 8.5: Results of RCMGoal on *HydrogenTank* and *HydrogenTankSmall*

for BioIK, but the condition that the cost function has to be smaller than 10^{-10} to accept the solution was replaced by enforcing a distance of no less than 1mm to the RCM.

The results show that both BioIK and RelaxedIK show substantial improvement when the RCMGoal is used compared to the baseline. Interestingly, BioIK seems to improve significantly more than RelaxedIK when the goal is used. A reason for this could be that the respective weight of the goal is different for BioIK and RelaxedIK and therefore the RCMGoal influences the overall cost differently. Still, it is evident that the RCMGoal is very promising and significantly improves the results on *HydrogenTank* and *HydrogenTankSmall*. For most of the other scenarios, for example *Shelf* or *TableObjects*, this approach is not usable because there is no clear RCM that can be used.

The function can also be changed such that only one link can be used to traverse the RCM. For that, only the argument of the min function is used as the objective function. As Figure 8.6 shows, this results in faster convergence. However, some points are not reached that were reached with the previous approach because it is not possible to reach them when the longest link is used for the RCM.



Duration	5ms	10ms	100ms	1000ms
BioIK	0.1%	1.3%	17.3%	68.9%
BioIK (RCMGoal single link)	**3.8%**	33.8%	94.0%	95.2%
BioIK (RCMGoal multiple links)	0.0%	1.2%	55.6%	98.6%
RelaxedIK	0.6%	1.1%	14.1%	48.4%
RelaxedIK (RCMGoal single link)	2.9%	**43.1%**	**94.3%**	96.1%
RelaxedIK (RCMGoal multiple links)	2.0%	16.6%	74.6%	**99.7%**

Figure 8.6: Comparison of different RCM goals on *HydrogenTank*. Best values are printed in bold.

PivotIK builds upon the same approach but uses a different method for optimization. BioIK is also able to reach the pose and keep the RCM much faster, but still many of the obtained configurations result in collisions.

The Python implementation had a similar speed on my computer[24] as on theirs (around 75ms). The C++ implementation seems to be considerably faster (they claim 0.75ms per solve). However, when using ELISE and the hydrogen tank goals (without collision checking) the solver is much slower, its solve speed reduces to an average of 741 ms per solve. Assuming the advantage of the C++ implementation scales linearly, it would be able to solve the problem in 7.41ms. While this result reaches the pose and keeps the RCM, it does not necessarily avoid all collisions. BioIK performance on ELISE was at 9.7ms with the RCMGoal, which is slower, but in the same order of magnitude. The C++ implementation of PivotIK is not published at the time of writing, so there is no way of verifying their results or testing the IK solver on ELISE. Therefore, the current implementation of PivotIK cannot be used to speed up the solving process, but a future C++ release could potentially change that. The difference to BioIK, however, is comparably small so that no significant speedup is expected from the C++ implementation.

8.6 Collision Point Distance

Since the analytical RCM objective described in the previous chapter does not work if the robot does not have straight links, a solution considering link geometries has to be found. To find the distance between a robot and a point in space, the collision checking library can be leveraged. A possible approach for this is to create a collision scene containing only the robot and a sphere at the RCM, disabling self-collisions and checking for collision distance.

In order to make sure to only encourage using the RCM with the correct link, the allowed collision matrix can be used. Collisions between all links can be allowed, only between the target link and the RCM object, they should be forbidden. Therefore, the collision distance is only calculated between these objects.

Concerning the implementation, a sphere of diameter 1cm was added to MoveIt's planning scene at the RCM point. The distance was calculated using the `distanceToCollision` method and used as input for the groove loss (parameters: $s = 0, c = 0.01, n = 1, r = 10, g = 2$).

When no ACM is used, i. e. any link can go through the RCM, the approach is significantly slower than the RCMGoal. For small timeouts, the solver also performs worse than the RelaxedIK solver without any additional goals. However, starting from 200ms, the goal starts to outperform the pose-only solver. A final score of approximately 75% is reached on *HydrogenTank* after the full 1000ms timeout. The collision point distance function without an ACM is therefore not a suitable objective function.

When the ACM is used, the solver performs very well, solving 97.5% of the targets. Surprisingly, this is more than the solve rate of the analytical RCM solver using the single link (96.1%). A reason for this could be that the solver can terminate slightly earlier than when the RCMGoal is used because the diameter of the collision object and the robot

[24] Intel Core i7-1365U (6 cores / 12 threads)

link make the objective function reach its minimum slightly earlier. The same reason could explain the worse performance when the opening of the fuel tank is smaller. There, every additional centimeter of deviation from the opening center increases the chance of collisions drastically.

On *HydrogenTank*, both solvers manage to find more than 50% of the solutions during the first solver iteration.

Table 8.5 shows the detailed time analysis of the solver. Calculating the collision distance between a sphere and a single link of the robot is very fast, taking only 1.15 microseconds. Therefore, much less of the calculation time is spent on collision checking and the solver can perform many calls to the objective function. As can be seen, the RCM metric reduces the average number of objective function calls when a solution is found from around 63k to 12k.

This result shows that collision checking and distance query is not generally slow but depends on the problem. Using a single collision object to enforce an RCM goal is a viable alternative to the analytical approach and therefore makes it possible to leverage the advantages of the RCM goal for robots with non-straight links. The differences between the

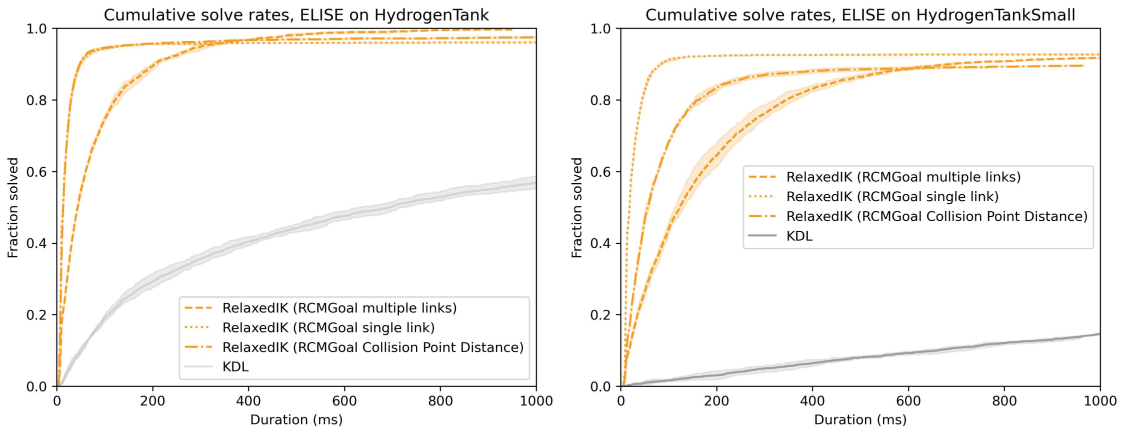


Figure 8.7: Comparison of RCM goals on *HydrogenTank* and *HydrogenTankSmall*

Solver	Successful	Total time	Calls to objective function	Time spent on collision checking	Time per collision check
RelaxedIK	✓	334.4ms	63129.9	×	×
	×	1001.2ms	206476.4	×	×
RelaxedIK with Collision Point Distance	✓	33.15ms	11863.8	12.7ms (22.4%)	1.15μs
	×	1000.2ms	516297.9	553.6ms (55.4%)	1.15μs

Table 8.5: Time spent on collision point distance in RelaxedIK when the ACM is used (averaged over the 1000 IK queries of *HydrogenTank*). Results are grouped by their query success.

analytical and numerical RCM goals with multiple links highlight the importance of fast objective function execution. Even though the objective function is good, as the analytical experiments show, not as many solutions are found because the function execution takes too much time. Solver iterations are independent of objective function execution times and provide a good measure of the potential of an objective function.

8.7 Line Objective

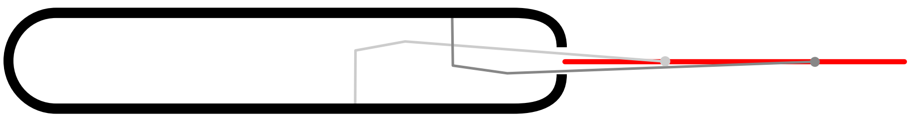


Figure 8.8: Illustration of the line objective, keeping a link (circle) on the red line

Another constraint that is highly specific to the problem of fuel tank inspection is to constrain the position of ELISE's first link, the longest bar, to a line. The line goes through the opening of the container and runs parallel to its length. Placing the base of ELISE's first link on this line, the robot should always pass through the opening without colliding with the container. The disadvantage of this method is that it only works for elongated objects and robots that have a long, rod-like link. It is also required to specify not only the opening point but also the direction in which the container extends.

The simplest formulation of this objective is the following, constraining a single point p of the robot to a line along n through p_l.

$$f_{\text{line}} = \|p_l - p_p\|$$

with

$$p_p = p - (n \cdot (p - p_l))n$$

This is also available as the LineGoal in BioIK 2.

However, this approach does not constrain the orientation of the link. This can be achieved by including the orientation as a separate goal. The BioIK OrientationGoal, for example, is implemented in the following way:

$$f_{\text{orientation}} = \min(\|q_l - q\|^2, \|q_l + q\|^2)$$

An alternative is to calculate the angle between the two quaternions, i. e.,

$$f_{\text{orientation}} = 2\arccos(|q \cdot q_l|)$$

According to [Rup+18], the first variant of the function leads to faster convergence as it is faster to compute and has a better gradient.

Duration	5ms	10ms	100ms	1000ms
BioIK (LineGoal)	6.7%	17.9%	96.1%	97.9%
RelaxedIK (LineGoal)	2.3%	20.1%	95.0%	97.4%

Table 8.6: Quantitative results of LineGoal on *HydrogenTank*

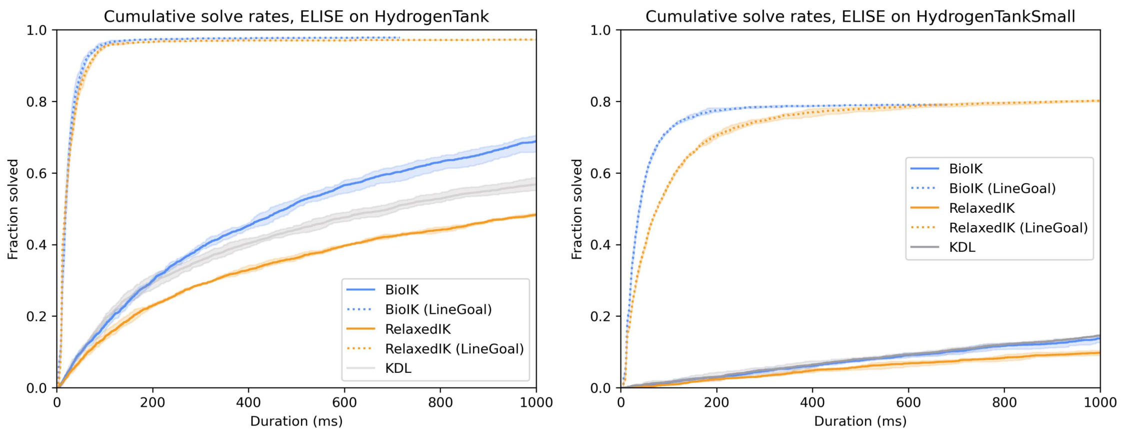


Figure 8.9: Results of the line objective on both fuel tanks

The only targets that are not found are those that are very close to the opening. Interestingly, KDL is able to find them. The results on *HydrogenTankSmall* shown in Figure 8.10 give a clearer view of the problem. Because of the link lengths and joint limits of ELISE, the targets can only be reached when the robot arm is lifted or lowered. Figure 8.11 shows a pose that was not achievable using the LineGoal.

To conclude, the LineGoal gives good results for a lot of targets, but it also constrains the link's position considerably. Its usability depends strongly on the geometries of the robot and the specifics of the problem because it does not represent an idea that is as intuitive as the RCM goal.

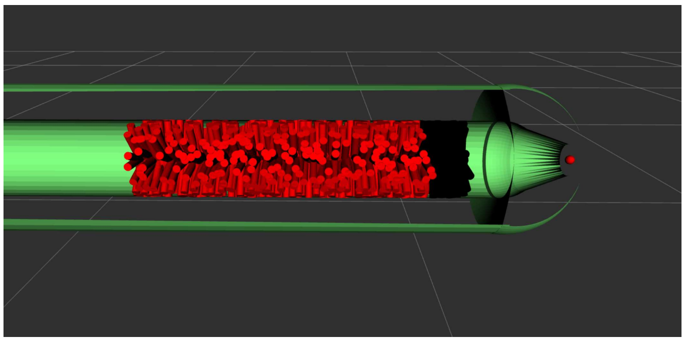


Figure 8.10: Cross section of *HydrogenTankSmall* with targets reached by RelaxedIK. Red arrows show reachable targets, black are not reached. The opening is on the right side, the line objective goes horizontally through the red point. BioIK results look similar.

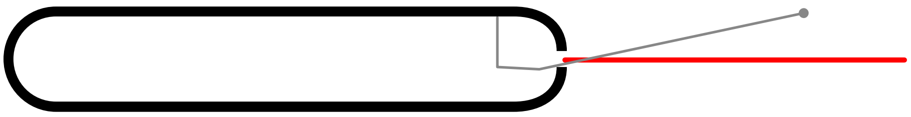


Figure 8.11: Example solution for a target that is unreachable when using the LineGoal. Due to link geometries and joint limits, the second joint of the endoscope is below the opening, requiring the first link to be tilted. The line that the root of the endoscope (circle) should be on is marked in red.

8.8 Alignment Objective

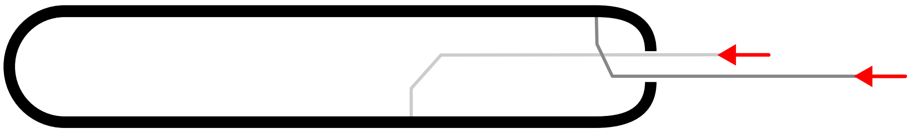


Figure 8.12: Illustration of the alignment objective, constraining the orientation of a link along an axis

The line objective constrains the position of a link of the robot. It may be desirable to instead constrain its direction. For this reason, the alignment objective can be used. It encourages a link to align with a given axis.

The cost function for the alignment objective is chosen as the distance between the unit vectors in the target and the current orientation. The vector distance is used instead of a quaternion difference because the rotation around the direction vector does not matter. This cost function is already available as the DirectionGoal in BioIK and was added to RelaxedIK. RelaxedIK uses the groove loss with parameters $s = 0, c = 0.1, n = 1, r = 10, g = 2$.

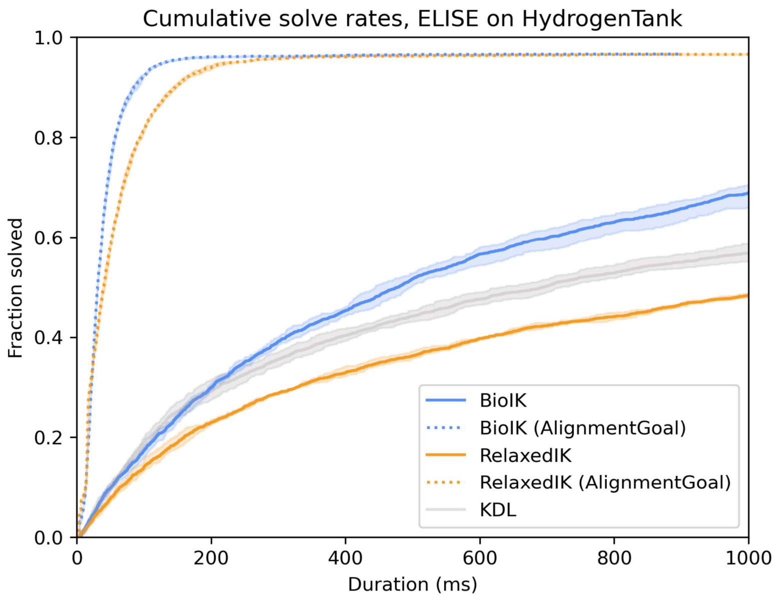


Figure 8.13: Results of the alignment objective on *HydrogenTank*. A comparison to the previous approaches can be found in Section 8.17.

Duration	5ms	10ms	100ms	1000ms
BioIK (AlignmentGoal)	3.1%	5.7%	92.4%	96.7%
RelaxedIK (AlignmentGoal)	5.4%	8.1%	81.5%	96.6%

Table 8.7: Quantitative results for AlignmentGoal on *HydrogenTank*

The results show that the alignment objective also works for *HydrogenTank*, but the performance is worse than for the line objective. The reason for this is that, like for the line objective, the link geometries and joint limits of ELISE do not make is possible to find solutions that fulfill the alignment criteria and do not collide. The example shown in Figure 8.11 also does not work for the alignment objective; the objective constrains the problem even so much that no solutions at all are found on *HydrogenTankSmall*. Therefore, use of the alignment objective is discouraged because it constricts the robot's solution space too much and depends highly on the problem geometries.

8.9 Look At Objective

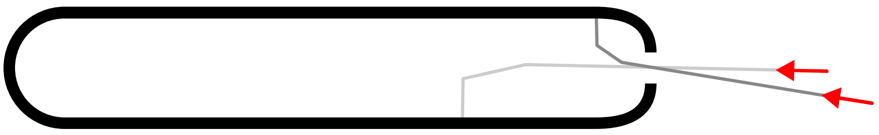


Figure 8.14: Illustration of the look at objective. The red arrows have to point towards the tank opening

Instead of using the alignment of the first endoscopic link, its direction toward the opening can be used as a metric. In principle, this is similar to the RCM objective because the link of the robot is straight and when it is pointed towards the tank opening, it will also go through the remote center of motion. This objective, however, also allows another link to go through the opening as long as the first link points towards it. The goal can be implemented as the dot product between the forward axis of the first link and the normalized vector between the link and the opening. This implementation is already available in BioIK as the LookAtGoal.

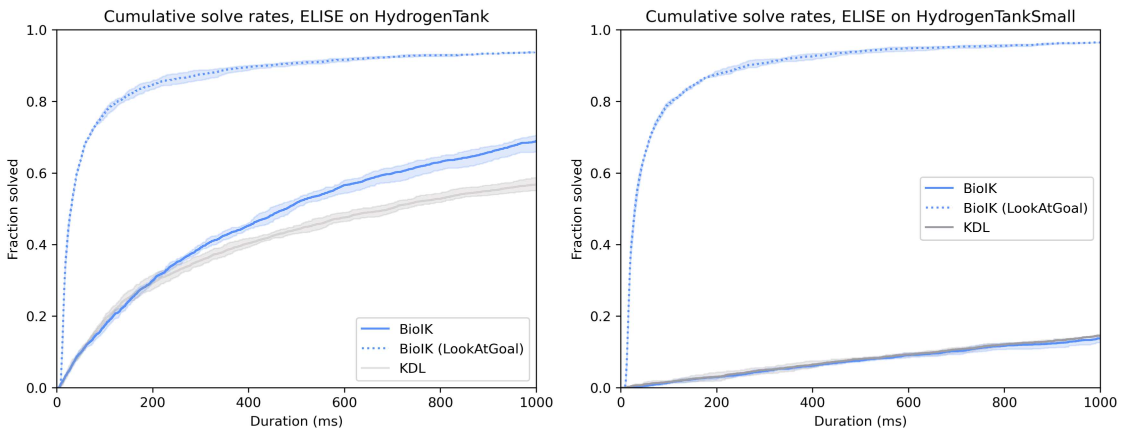


Figure 8.15: Results of the look at objective with BioIK on *HydrogenTank* and *HydrogenTankSmall*. A comparison to the previous approaches can be found in Section 8.17.

As can be seen, the goal also leads to a fast convergence, but not all points are reached. In contrast to the previous approaches, the solve rate on *HydrogenTankSmall* is also very good, with over 60% solve rate within the first solver iteration.

8.10 Kinematic Decoupling

An approach that works fundamentally different than the previous, cost-function-based approaches is to split the robot into its endoscopic (distal) and non-endoscopic (proximal) part. The endoscopic part is the part of the robot that enters the constrained environment. For the setup used in this thesis, ELISE is the distal part and the UR10 arm is the proximal part. With this partition, the inverse kinematics problem can be solved in two steps (see Figure 8.16). First, a solution for the endoscopic part is searched. This can be any solution for the joints of the endoscopic part that starts from the target pose and does not collide, optimally also reaching out of the constrained environment to avoid collisions of the proximal part. In the next step, inverse kinematics for the non-endoscopic part is solved. Here, the proximal part's end effector has to reach the pose of the distal part's base link. In this process, collisions do not have to be considered because the solution of the proximal part is outside of the constrained environment. Both partial solutions are then merged to obtain the solution for the full robot. This way, a solver using kinematic decoupling solves the inverse kinematics problem by splitting it into two tasks, one with few degrees of freedom and many potential collisions, the other with few collisions and more degrees of freedom.

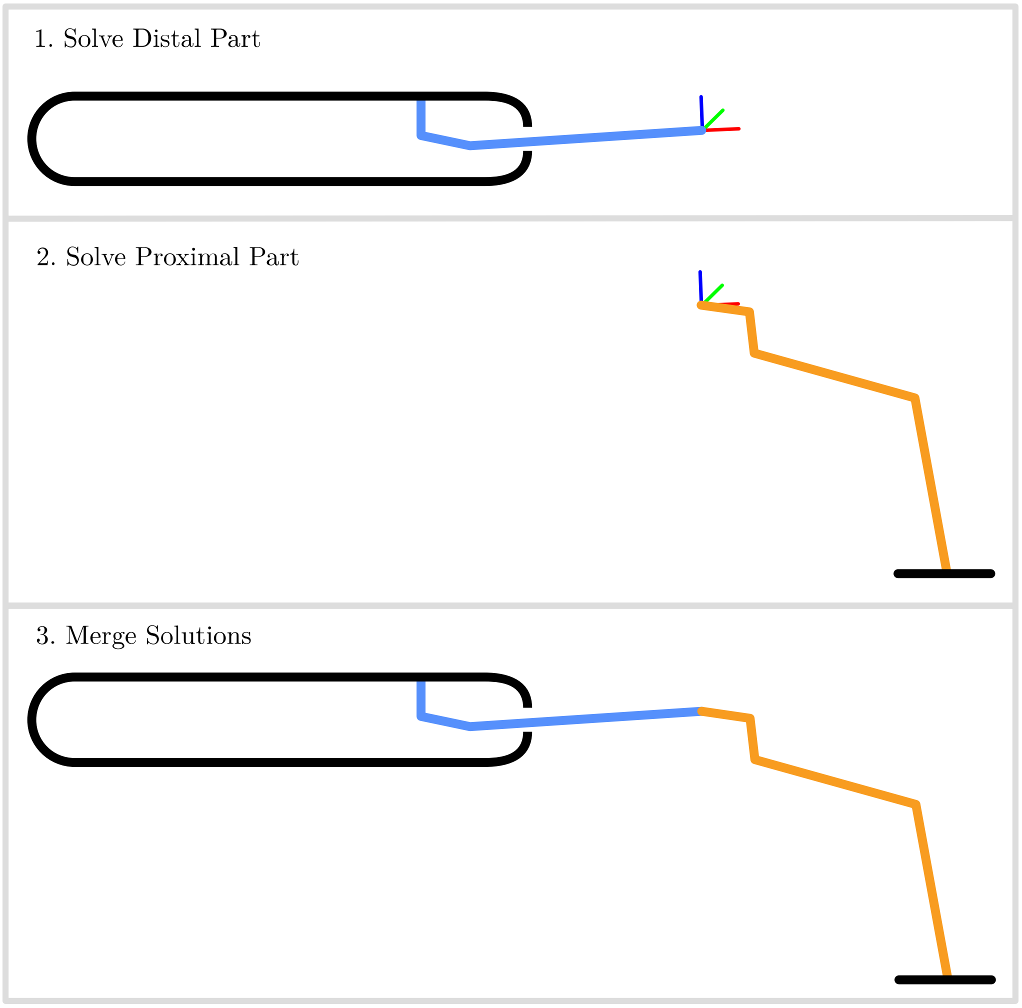


Figure 8.16: Illustration of kinematic decoupling

To achieve kinematic decoupling, the URDF (see Section 2.7) of the endoscopic part has to be inverted, i. e., a URDF has to be found that represents the inverted transform of the end effector while using the same joint configurations as the original robot. This can be achieved in the following way:

$$T_{\text{ee}} = T_1 T_2 T_3 = T_{1_{\text{origin}}} T_{1_{\text{joint}}} T_{2_{\text{origin}}} T_{2_{\text{joint}}} T_{3_{\text{origin}}} T_{3_{\text{joint}}}$$
$$T_{\text{ee}}^{-1} = T_3^{-1} T_2^{-1} T_1^{-1} = T_{3_{\text{joint}}}^{-1} T_{3_{\text{origin}}}^{-1} T_{2_{\text{joint}}}^{-1} T_{2_{\text{origin}}}^{-1} T_{1_{\text{joint}}}^{-1} T_{1_{\text{origin}}}^{-1}$$

This inverted transform can not be directly translated back to a URDF because it starts with a joint transformation and ends with an origin transform. Therefore, new parts of joints have to be added as identity transforms I at the start and end of the transform:

$$\begin{aligned} T_{\text{ee}}^{-1} &= I T_{3_{\text{joint}}}^{-1} T_{3_{\text{origin}}}^{-1} T_{2_{\text{joint}}}^{-1} T_{2_{\text{origin}}}^{-1} T_{1_{\text{joint}}}^{-1} T_{1_{\text{origin}}}^{-1} I \\ &= T_3' T_2' T_1' T_0' \end{aligned}$$

with

$$\begin{aligned} T_3' &= I T_{3_{\text{joint}}}^{-1} & T_2' &= T_{3_{\text{origin}}}^{-1} T_{2_{\text{joint}}}^{-1} \\ T_1' &= T_{2_{\text{origin}}}^{-1} T_{1_{\text{joint}}}^{-1} & T_0' &= T_{1_{\text{origin}}}^{-1} I \end{aligned}$$

T_0' is therefore a fixed joint, the other joints take the configuration values from their corresponding joints in the original URDF. The complete chain then reproduces the original kinematic chain.

This URDF inversion was done manually for ELISE to evaluate the use of kinematic decoupling. Automating the process is left for future work.

The inverted URDF is used as input for the distal IK solver. Any target or cost function given to the IK solver also has to be transformed to be expressed relative to the original target, which is now the base of the inverted distal part. Therefore, when T_p is the target, any pose, position, or orientation given to additional goals has to be multiplied with T_p^{-1} to perform this transform.

For solving the endoscopic part, the question remains which cost function should be used to help the solver steer toward the opening of the container. Different possibilities emerge. First, the penetration depth is considered. This approach does not work well because while the depth is non-zero in collision cases and zero in non-collision cases, it does not provide a clear gradient, possibly because it is approximated in FCL. Maximizing the distance to collision does not work either because it is always zero in collision cases and therefore does not have a gradient. Another idea is to maximize the distance between the contact point and the target, making the solver try to move the collision as far away from the target as possible. This approach can indeed lead to finding the way out of the constrained environment when the way out, e. g., the tank opening, is at the end of the object. However, it does not work in general because the opening does not always coincide with the point that is the farthest away. Since ELISE only has three degrees of freedom, it might also be possible to perform a grid search in configuration space. However, this approach is probably not efficient enough to be applicable in high-collision environments like the fuel tanks.

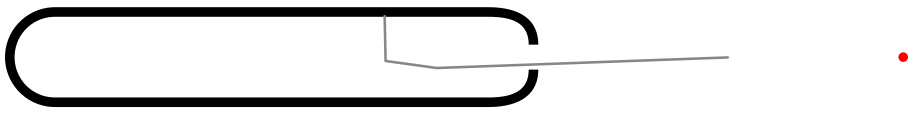


Figure 8.17: Illustration of the distal solver goal, the end of the distal part should be as close to the red point as possible.

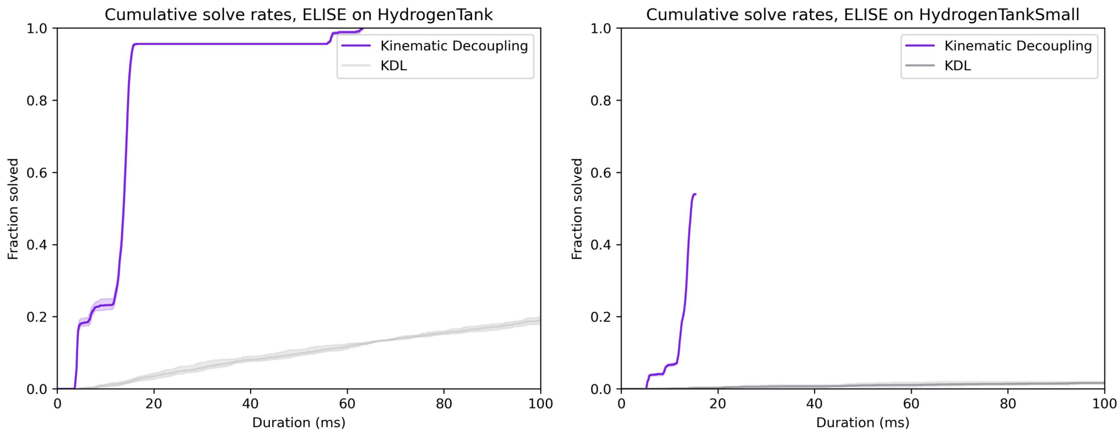


Figure 8.18: Results of kinematic decoupling on *HydrogenTank* and *HydrogenTankSmall*

When only the fuel tank scenarios are considered, some of the objectives from the previous sections can be used as well. The problem with many of these objectives is that they are much less clear when applied to the distal part only. For example, the LookAtGoal might find solutions that put the base link on the opposite side of the tank, making it unreachable for the proximal part.

A very simple alternative to these goals is to use a cost function optimizing for the position of the base link and giving a target that encourages the base link to point in the direction of the opening (see Figure 8.17). When the IK solver is set to solve the problem approximately, the end of the endoscopic part will be as close to this point as possible, resulting in it going though the tank opening. Since the solver for the distal part cannot solve the problem perfectly, a timeout has to be set. 1ms was found to be sufficient to find a solution. The proximal solver then uses the remaining time. The solver for the distal part will always find the same solution because the endoscopic part of the robot has only three degrees of freedom, leaving no redundancy for the solution. Therefore, the resampling strategy from Section 8.1 is not applicable and only a single solver iteration has to be considered.

This approach manages to solve 100% of the targets in the first iterations on *HydrogenTank*. On *HydrogenTankSmall*, it only manages to find 54% because of the problem mentioned in Section 8.7. Figure 8.18 shows the results. The presented implementation is using BioIK for both the proximal and the distal solver.

Finding additional goals that work well with kinematic decoupling is left for future work.

8.11 Optimization Solvers

The RelaxedIK implementation for C++, described in Section 7.2, uses nlopt-cpp as the optimization backend. It supports a variety of solvers, some of which are local, some are global, some use a gradient, others do not. In this thesis, it is interesting to find a solver which works best for the use case of inverse kinematics. As the fitness landscape is quite complex, I suspect that a global solver works better than a local solver, and that a gradient should be used in order to guide the solver into the right direction after a potential global minimum was found.

Preliminary experiments on the *Random* scenario show that the global solvers perform quite badly. While some of them find solutions that are in proximity of the targets, they often use up the full timeout for finding them. Subsequent local optimization to improve the result often leads to success but requires more additional time. The best global solver is StoGO, a gradient-based global optimizer that divides the search space into rectangles that are then searched with a local optimization solver. The other global solvers, i. e., DIRECT, AGS, ISRES, and ESCH do not give better results. The local solvers generally provide better results, with PRAXIS and Nelder-Mead giving the best results for gradient-free solvers. The best-performing solver is SLSQP, other gradient-based solvers, such as LBFGS, VAR, and TNEWTON also perform well, but worse than PRAXIS and Nelder-Mead. The BOBYQA and SBPLX algorithms require a little more time, the COBYLA, MMA, and CCSAQ algorithms even take considerably longer. They are therefore excluded.

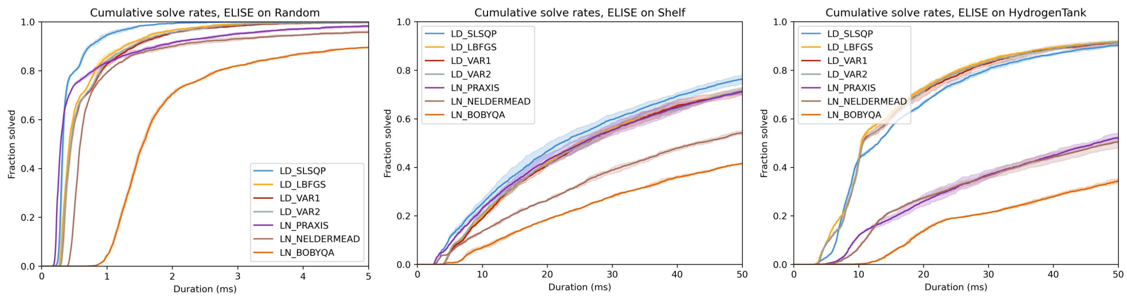


Figure 8.19: Different optimization solvers on three problems. Left to right: *Random* scenario, *Shelf* scenario, *HydrogenTank* with the RCMGoal ACM objective.

Figure 8.19 shows evaluations of the different local solvers on three different problems. For the *Random* scenario, the SLSQP solver performs best and most of the other solvers show similar results, except BOBYQA, which is slower. The *Shelf* scenario shows similar results. The third panel shows the *HydrogenTank* scenario with the RCM objective using the ACM. Here, a difference between gradient-free and gradient-based solvers becomes the most clear. In addition, SLSQP is slightly outperformed by the other gradient-based solvers.

The analysis shows that the SLSQP solver is a good choice for the inverse kinematics solver. Its speed and accuracy mostly surpass the other solvers. Lack of global solving capabilities seems to be compensated by the random restarts.

8.12 Seeding

MoveIt kinematics solvers accept, in addition to the target pose, a seed state. This seed state should be used to initialize the solver and find solutions that are close to the seed. Normally, it is set to the current state of the robot. An exemplary use case is to track a trajectory where new solutions should always be close to previous solutions. Of course, this feature can also be used to help the IK solver find solutions that are hard to reach. For example, a solution that already places the robot's end effector inside of a container could be used as a seed to find more solutions inside this container. The seed can either be supplied manually or be generated from the IK solver itself, e. g., by providing the solver with more time for the first solve than for subsequent solves using the generated seed.

In this section, the influence of a seed is evaluated. For this, only the *HydrogenTank* and *HydrogenTankSmall* scenarios are selected because for them, the seed configuration is straightforward. It should be a configuration where the endoscopic part of the robot is inserted into the object. Seed for the scenarios were supplied manually.

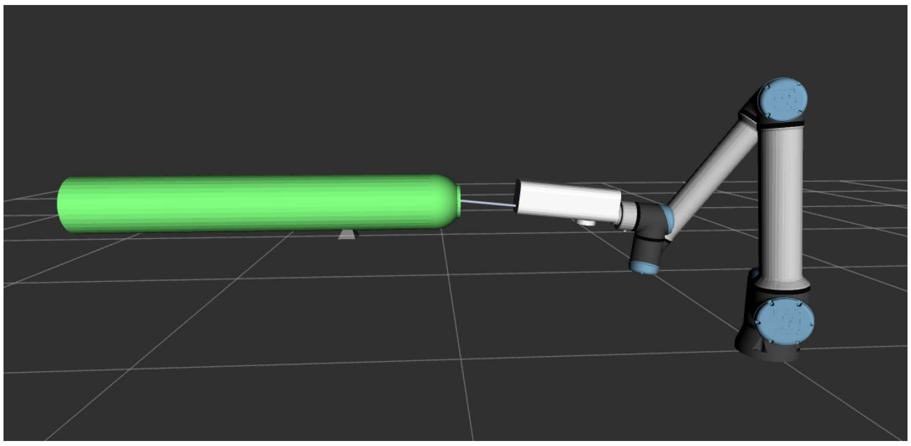


Figure 8.20: Seed position used for *HydrogenTank*

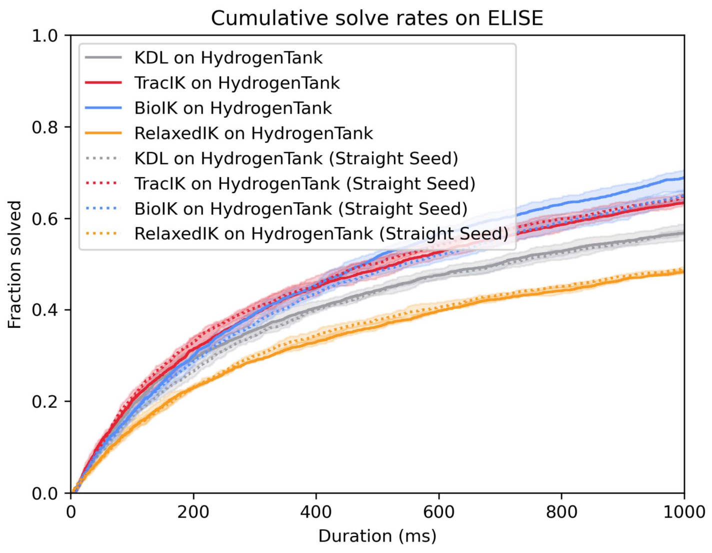


Figure 8.21: Solver performance when a seed is used on *HydrogenTank*. There is no significant difference between seed and no seed.

Solver	No Seed	Straight Seed	Seed pointing to target
KDL	0.0%	0.0%	48.2%
TracIK	0.0%	0.0%	44.6%
BioIK	0.0%	0.1%	31.4%
RelaxedIK	0.6%	1.0%	10.1%

Table 8.8: Success rates for the first iteration when a seed is used on *HydrogenTank*. The straight seed extends the endoscopic joints fully, the target seed uses a configuration for a single target as seed.

The results show a surprisingly small influence of the seed for all solvers. After the first call to the solution callback, it makes sense that the seed no longer influences the solution because all seed information is lost when rejection sampling is performed. Therefore, if the first solution found was not collision-free, the behavior is the same as without a seed. For the first call, the reason could be that the endoscopic part of the robot was positioned straight inside the the tank. Reaching a target therefore requires a large movement of the endoscopic joints while any movement of the arm joints should be avoided. This specificity will not be fulfilled by any solver that does not account for the collision object.

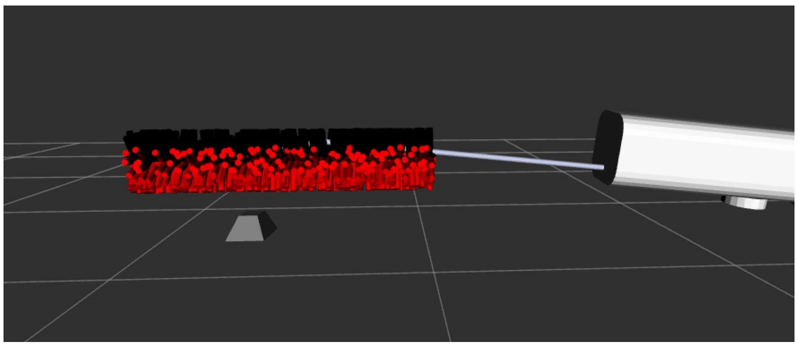


Figure 8.22: Targets reached by KDL with a seed within the first solver iteration. The seed configuration is shown by the robot model

When a seed that already reaches a target is used, this hypothesis could be tested because the surrounding targets would be found within the first solver iteration. Using a solved target pose as seed results in significantly better results. KDL and TracIK then find almost 50% of the solutions within the first iteration. Table 8.8 shows a comparison of the two seeds and no seed. For the deterministic solvers, the seed configuration has to be randomized after the first solution failed the callback. For BioIK, as it is randomized, the seed could be used multiple times with different results. For that, I implemented a version that does not randomize its configuration after the solution callback failed, but instead reuses the seed. The results are shown in Figure 8.23. For the normal solver, the seed brings a significant initial increase in reached targets, targets that could not be solved within the first iteration are not reached with a higher probability later on. When the seed is kept, the next iterations still discover more solutions. However, the overall capabilities of exploration are limited and the normal solver eventually surpasses the one that keeps the seed.

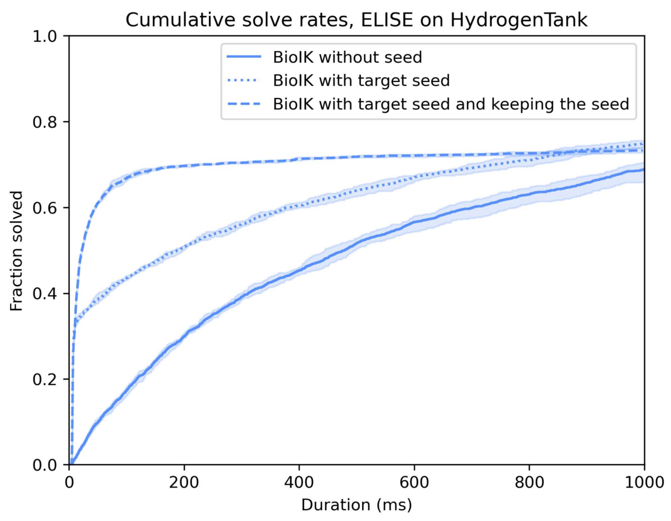


Figure 8.23: Seed variants for BioIK on *HydrogenTank*: no seed, seed pointing to a target, seed pointing to a target without random re-initialization

For *HydrogenTankSmall*, when a seed is used, the KDL solver can solve 26.4% of the targets within the first iteration. While this is less than on *HydrogenTank*, it is still a comparably high number of solutions that are found with a relatively easy approach.

Combining the seed with other approaches developed in this thesis results in more solutions being found within the first solver iteration. However, their results do not improve significantly because a lot of these solutions would have been found within the first few iterations anyway.

A different approach to handling a seed would be to use joint weights that penalize a large movement of some joints. In this case, movement of the endoscopic joints and wrist_3_joint, the joint that controls the rotation around the endoscopic axis, could be allowed, while movement of the other arm joints is reduced.

To conclude, a well-selected initial seed can improve the solve rates significantly. It would be a viable approach to solve for a target using a high timeout or more complex goals and then use this as seed for another solver. KDL, for example, manages to solve around 50% of the targets collision-free within a single iteration of the solver when a good seed is given.

8.13 Surface Inspection

In surface inspection tasks, the end effector of the robot does not need to touch its target pose. Instead, a camera is mounted on the end effector and takes pictures or videos of the surface which is located at a certain distance from the camera. While this is technically identical to adding a virtual end effector that has to touch the surface to the tip of the robotic arm, a camera has more tolerances toward inaccuracies than a gripper. Therefore, the pose of the camera can be rated using a value function that gives the highest value to the perfect camera position but allows for discrepancies in the distance and angle toward the target. This can potentially change IK performance because a different measure of performance is introduced. In this section, approaches to this cost function approach are implemented and evaluated using custom goals for RelaxedIK.

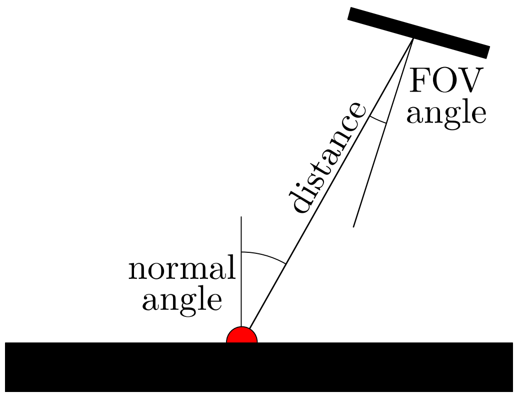


Figure 8.24: Illustration of inspection criteria. The camera on the top right is inspecting the red target.

For surface inspection, three complementary criteria can be used: The distance to the target is the distance from the robot's end effector with the camera to the point that should be inspected. A good value for it depends on the focal distance of the camera. The FOV angle is the angular distance between the center of the camera's field of view and the target object. Keeping it small ensures that the object is centered in the image. The normal angle is the angle between the normal of the plane the target is on and the camera. Small values correspond to a camera that is positioned above the target. In addition, these criteria may be extended with line-of-sight checking, i. e., checking whether an object is between the sensor and the target. Since this requires potentially expensive collision checking operations, it can be replaced with a goal limiting the maximum distance to the target.

The criteria for scanning are implemented in RelaxedIK using the swamp loss with parameters $a_1 = 1, a_2 = 0.1, n = 8$. The wall values are 0.0 and 0.02 for the distance, and -0.2 to 0.2 for the angles (approximately 11°). The termination criterion for RelaxedIK is changed to enforce distance, FOV angle, and normal angle values within these walls. The idea of these cost functions is that solutions will be found quicker and more of them pass the collision check. In combination with other goals, e. g., the RCMGoal, a broader cost function could allow more solutions that fulfill these other goals.

Figure 8.25 shows the results of the evaluations. For the *Random* scenario, the scan goal is slower than the default goal, which can be explained by the fact that its calculation is more expensive. This hypothesis is supported by the more similar number of solver iterations. For the *HydrogenTank* scenario, the performance of the ScanGoal exceeds the other goals. When no additional goal is used, the ScanGoal is faster in terms of time and iterations than the default goal. The reason for this is probably that it leaves more freedom to the solver and therefore favors more exploration or makes collision-free solutions possible that were not possible with the default goal. When it is combined with the RCMGoal, the evaluations show similar results in terms of time, but a better performance in terms of solver iterations. This can be explained by the fact that the ScanGoal has a wider range of poses where it is minimal, leaving more freedom to optimization of the RCMGoal, but takes more time to calculate. On *HydrogenTankSmall* (c.f. Appendix A14), the results are similar to *HydrogenTank* but with a larger advantage for the ScanGoal.

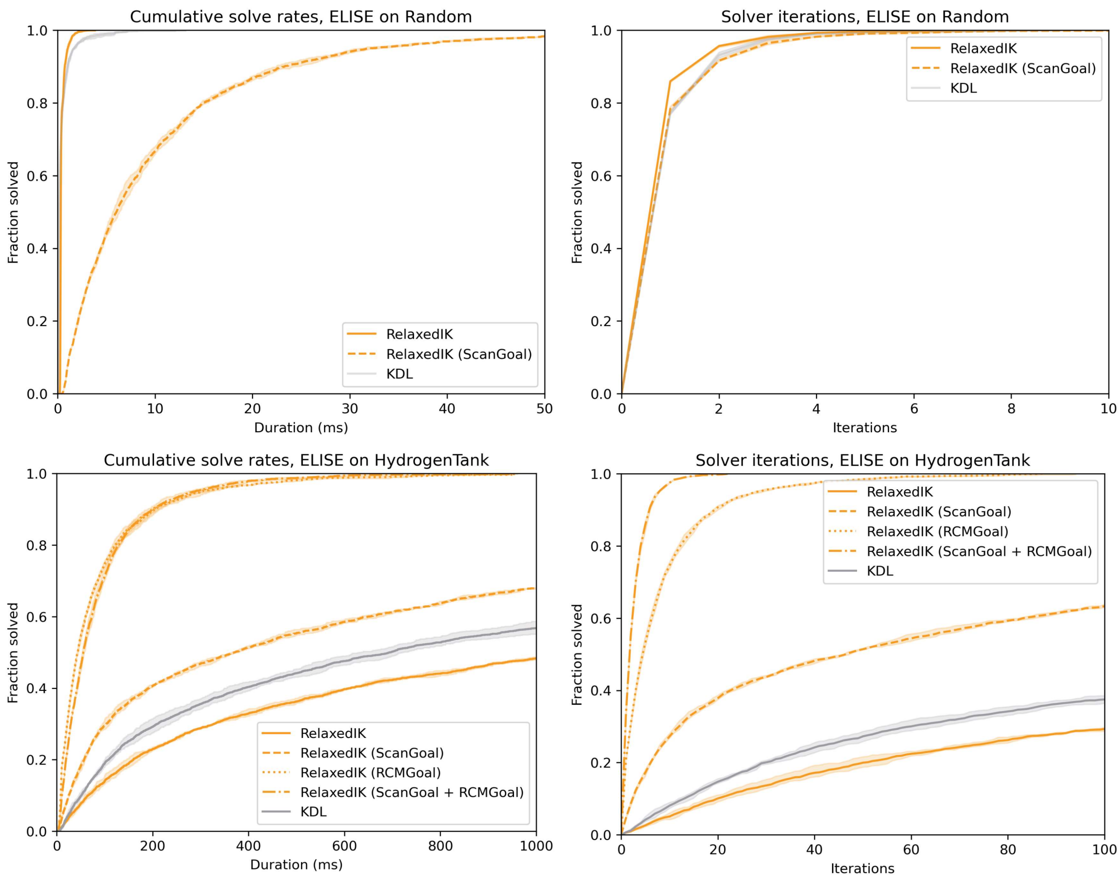


Figure 8.25: RelaxedIK with ScanGoal on *Random* and *HydrogenTank*

Using the criteria in BioIK without a swamp loss did not result in finding better solutions. The explanation for this is straightforward: While in theory, some deviations from the target are possible with a slight increase of cost, this is already the case for the normal cost function used to reach target poses. The algorithm will still always find a solution that fits the best, which is at the exactly specified distance and angles. In BioIK, the threshold of 10^{-10} for the cost function also has to be met, leaving little freedom for possible solutions. The resulting solution is then checked for collisions and will fail approximately with the same rate as when target poses are used directly.

In conclusion, the usage of inverse-kinematics solvers for complex cost functions is effective as long as sensible termination criteria are used. Additional freedoms introduced by these cost functions can increase solve speed and combinations of different goals can combine their advantages. The creation of such goals also highlights the advantage of cost-function-based inverse kinematics solvers that complex goals, constraints, and freedoms can be modeled and used to find solutions in more general tasks than simple pose goals can cover.

8.14 BioIK Modes

In these evaluations, some of the existing BioIK modes are compared. This aims to reproduce the results of [Rup17], where BioIK 2 was developed. The results show that the jac8 mode, an implementation of the KDL solver using eight threads, finds most solutions

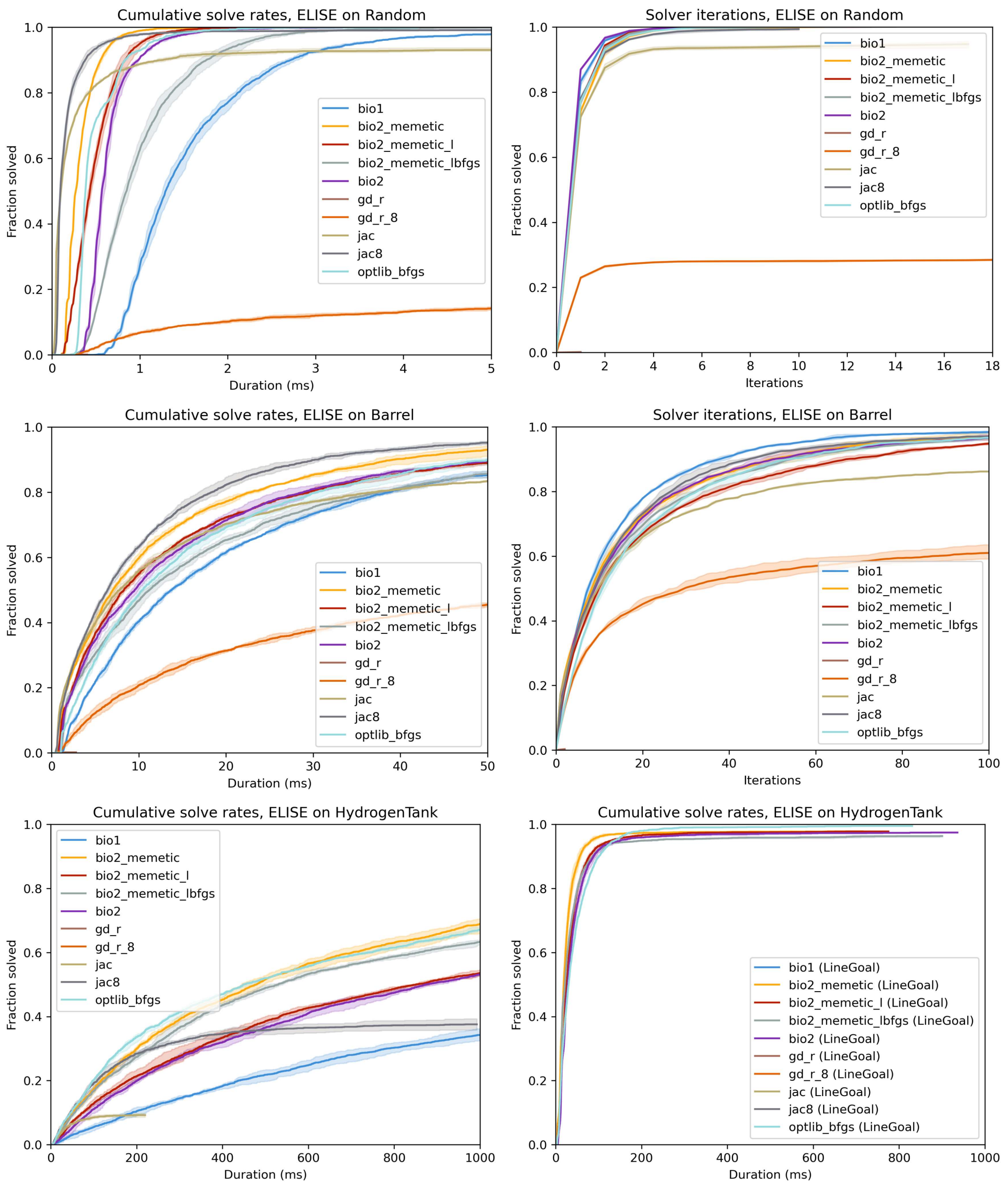


Figure 8.26: Comparison of different modes of BioIK 2. Top two rows show solve rates over time and over solution callback calls for *Random* and *Barrel*, respectively. The bottom row shows solve rates over time for *HydrogenTank*, on the left using rejection sampling, on the right using the line objective.

the fastest for simple scenarios such as *Random* or *Barrel*. However, it does not find all solutions, indicating that it lacks features for global exploration. These observations match the expectations because Jacobian-based algorithms do not have any means for exploration while they are very fast. The bio2_memetic mode is the second fastest mode and finds all solutions. An interesting remark is that bio1 is one of the slowest solvers but requires the fewest iterations to find the solutions. This indicates that the algorithm might be better at exploring the solution space than the other algorithms but is so much less performant than the other algorithms that it falls behind there.

Surprisingly, both gradient descent modes performed very poorly. Since gradient descent lacks features for global exploration, these algorithms were not expected to perform extremely well. Still, more solutions should have been found, especially by the single-threaded variant. I suspect that these algorithms are not implemented correctly. A comparison to PickIK's local mode, which is in theory a verbatim copy of BioIK's gradient-descent mode, shows that the algorithm is capable of solving 48% of the targets within 5ms.

For the *HydrogenTank* scenario, bio2_memetic is the best solver when the full timeout is used. The optlib_bfgs mode performs comparably well. When the line objective is used instead of simple rejection sampling, the bio2_memetic solver is still the fastest solver to find many solutions, however, it is eventually surpassed by the optlib_bfgs solver.

To conclude, the bio2_memetic solver is a good general-purpose solver. While it can be surpassed by Jacobian-based solvers on simple problems and optimization-based solvers on more complex problems, it consistently performs well in all tested areas.

8.15 BioIK Cost Functions

In order to evaluate how well different cost function goals work with the BioIK 2 implementation, it is relevant to see how much overhead cost function execution induces. First of all, a distinction between two types of cost functions has to be made. The BioIK-internal cost functions use BioIK's own implementation of forward kinematics, a `bio_ik::Frame`. MoveIt cost functions use a `moveit::core::RobotState`. On this robot state, MoveIt's functions such as collision checking can be executed.

Most cost functions mentioned in the previous sections, such as the line, alignment, and RCM objectives, were implemented as BioIK-internal cost function. Those that require collision checking and therefore need MoveIt's RobotState, are implemented as MoveIt cost functions.

To execute MoveIt cost functions, BioIK therefore first has to create or update a RobotState object. The time delay this adds was evaluated by providing BioIK with an `IKCostFnGoal` that contains a cost function returning a cost of zero. The execution of this cost function only takes negligible time and does not change the outcome of the algorithm, therefore all performance changes must come from the overhead induced by the cost function.

The setup was evaluated on the *Random* scenario.

Duration	5ms	10ms	100ms	1000ms	Mean
BioIK	100.0%	100.0%	100.0%	100.0%	0.335ms
BioIK (IKCostFn)	0.0%	0.0%	80.8%	100.0%	73.572ms

Iterations	1	2	3	10	Mean
BioIK	74.1%	92.2%	97.2%	100.0%	1.386 its
BioIK (IKCostFn)	74.4%	93.3%	98.0%	100.0%	1.356 its

Table 8.9: Results of BioIK with and without an IKCostFn on *Random*

Table 8.9 shows that the solver using the IKCostFn is significantly slower than the solver that is not using it. The iterations counter, however, shows very similar results, indicating that indeed the execution of the function is expensive, not the solver is performing worse. This confirms the hypothesis that IK cost functions in BioIK are very expensive, slowing the solver down so much that a target that was reached in under 5 ms without a cost function requires more than 20 times as much time with it. The reason is that BioIK has to create and update a RobotState object every time the cost function is called, which is several thousand times. The consequence of this result is that MoveIt IK cost functions are not usable with BioIK if a high solve speed is to be achieved.

The RelaxedIK solver developed in this thesis does not have this problem, as it works with MoveIt's RobotState internally (Appendix A13)

8.16 BioIK Parameters

For BioIK parameter optimization, EBIKE is employed, which uses the Optuna library for parameter optimization. The parameters that were described in Section 7.3 are optimized. Table 8.10 shows the ranges of these parameters used in the optimization.

Parameter	**Lower Limit**	**Upper Limit**	**Default Value**
`population_size`	1	20	2
`child_count`	0	100	16
`elite_count`	0	`population_size`	1
`threads`	1	8	4
`species_count`	1	20	2
`memetic_evolution_gens`	1	50	8
`memetic_opt_gens`	1	50	8

Table 8.10: BioIK parameter ranges used for parameter optimization

For the optimization, three different problems are selected. First, the optimization is done for the *Random* scenario. This should verify the selection of parameters made during the implementation of BioIK 2. Second, the *TableObjects* scenario is optimized. This scenario represents a medium level of collisions and should evaluate which parameters can be used to better treat situations where some exploration is needed to find suitable situations. Finally, the LineGoal on *HydrogenTank* was optimized because this goal performed the best and it would be interesting to see how the parameters could be changed to account for the function.

The optimization was run for 200 iterations. For ranking the evaluation trials, the mean of all IK times was calculated, where unreached targets were counted as the timeout of 1s.

$$\text{score} = \frac{1}{|T|}\sum_{t\in T}(\text{if reached}(t) \text{ then time}(t) \text{ else } 1)$$

For all scenarios, the results could be improved using the parameter optimization.

8.16.1 Random

The original parameters gave a score of 0.541ms, after optimization, a score of 0.269ms could be attained. This is a reduction of 50%.

Table 8.11 shows the ten best parameter sets from the optimization and the default parameters at the bottom. All of the best parameter sets are relatively similar. Clearly, they focus on optimization rather than evolution, which can be seen from the low number of evolution generations (0-1) compared to the high number of optimization generations (18-21). The population size was changed to 1, which makes sense as a second individual is not required when the main focus is on optimization. The relatively high child count (8-30) is probably used for further exploration, but the range of values indicates that the role of this parameter is minor. Optuna also ranks parameter importance, the result of which is that the number of optimization generations is by far the most important parameter, followed by child count and population size. It is interesting to note that all of the best parameter sets use a thread count of six. Apparently, this thread count maximizes the balance between result improvement and parallelization overhead.

Score (ms)	child_ count	elite_ count	memetic_ evolution_ gens	memetic_ opt_gens	population_ size	species_ count	threads
0.269	25	1	1	21	1	2	6
0.272	28	1	0	19	1	2	6
0.276	9	1	0	18	1	2	6
0.283	8	1	0	19	1	2	6
0.285	25	1	1	21	1	2	6
0.290	18	1	1	21	1	2	6
0.291	30	1	1	21	1	2	6
0.298	13	1	1	18	1	2	6
0.300	22	1	1	20	1	2	6
0.310	19	1	1	21	1	2	6
0.541	16	1	8	8	2	2	4

Table 8.11: Best ten parameter sets and default parameters for BioIK on *Random*

Score (ms)	child_ count	elite_ count	memetic_ evolution_ gens	memetic_ opt_gens	population_ size	species_ count	threads
4.98	2	1	3	15	3	2	7
5.24	2	1	1	15	4	2	8
5.25	38	1	1	15	4	2	8
5.26	46	1	1	19	4	2	8
5.29	2	1	3	16	3	2	6
5.32	44	1	2	17	4	2	8
5.35	43	1	1	17	4	2	8
5.48	2	1	1	17	3	2	7
5.51	46	1	2	20	4	2	8
5.56	45	1	1	21	4	2	8
6.17	16	1	8	8	2	2	4

Table 8.12: Best ten parameter sets and default parameters for BioIK on *TableObjects*

8.16.2 TableObjects

For the *TableObjects* scenario, an improvement by 20%, from 6.17ms to 4.98ms could be achieved.

For this experiment, the results differ from the results for *Random* in several ways. First, the trade-off between optimization and evolution is slightly different, still focusing on optimization but allowing for more evolution. This can be seen from the higher number of evolutionary generations and lower number of optimization generations, as well as the higher population size (3-4). The number of threads has often been increased to seven or eight, once again highlighting the exploration that can be done when multiple species are run in parallel. The child count still has a high range, indicating that it does not highly influence the results. According to Optuna, the most important parameter by far was the number of optimization generations. However, in total, the improvement compared to the original parameters is not substantial.

8.16.3 HydrogenTank

In the *HydrogenTank* scenario, parameter optimization could improve the time from 71.9ms to 51.8ms, a reduction by 28%.

In the *HydrogenTank* optimization procedure, the trade-off between evolution and optimization has been shifted even more. While more optimization generations are still favorable, there are now around the double of the evolution generations. Most notable, the number of elites, which was consistently 1 in the previous optimizations, has risen to mostly 2, with some larger exceptions. The population count was increased again, to 3-6 individuals, and the child count was drastically increased to approximately 80 children. Only the species count is consistently at 2 in all evaluations. According to the hyperparameter importance ranking, the child count is the most important parameter, followed by the species count and the number of evolution generations.

Score (ms)	child_ count	elite_ count	memetic_ evolution_ gens	memetic_ opt_gens	population_ size	species_ count	threads
51.8	75	2	9	21	4	2	7
51.9	81	2	9	21	5	2	7
54.0	89	2	8	13	5	2	7
54.2	84	2	8	11	3	2	7
54.2	68	10	5	29	6	2	5
54.8	77	3	8	21	4	2	7
54.9	87	2	9	11	4	2	8
54.9	83	2	9	17	4	2	7
55.0	8	2	8	14	4	2	7
55.0	81	2	9	21	6	2	7
71.9	16	1	8	8	2	2	4

Table 8.13: Best ten parameter sets and default parameters for BioIK with LineGoal on *HydrogenTank*

The results of BioIK parameter optimization show that while no enormous performance increase is possible, the parameter selection has an influence on the performance of the IK solver. Throughout all experiments, the optimal number of optimization generations is higher than the number of evolution generation. This change can probably be taken as a default. Additionally, the population size and child count can be increased.

8.17 Results

Table 8.14 lists the results of all experiments. It can be seen that, depending on the error metric, different approaches perform better. The best approaches are kinematic decoupling and using the target seed. Other approaches that work very well are using a remote center of motion (RCMGoal multiple links, Collision point distance), the AlignmentGoal, the LineGoal, and the LookAtGoal. Figure 8.27 compares the best approaches that are based on geometric considerations. The same table and figure for *HydrogenTankSmall* can be found in Appendix A15. There, the kinematic decoupling and target seed approaches have a much lower success rate. Additionally, the AlignmentGoal does not work at all and the LookAtGoal outperforms the LineGoal.

Approach	**Timeout**			**Iterations**	
	10ms	**100ms**	**1000ms**	**1**	**10**
Baseline (KDL)	0.9%	18.9%	56.8%	0.0%	7.2%
RelaxedIK	1.1%	14.1%	48.4%	0.6%	5.1%
Rejection Sampling (BioIK)	1.3%	17.3%	68.9%	0.0%	6.4%
Collision Distance (RelaxedIK)	0.0%	0.0%	1.1%	0.6%[25]	4.1%[25]
Penetration Depth (RelaxedIK)	0.0%	3.6%	25.3%	2.9%[25]	22.0%[25]
RCMGoal single link (RelaxedIK)	43.1%	94.3%	96.1%	51.6%	95.7%
RCMGoal multiple links (RelaxedIK)	16.6%	74.6%	99.7%	20.8%	74.5%
Collision point distance (RelaxedIK)	34.7%	94.5%	97.5%	54.9%	97.3%
LineGoal (BioIK)	17.9%	96.1%	97.9%	27.0%	92.6%
AlignmentGoal (BioIK)	5.7%	92.4%	96.7%	10.9%	87.8%
LookAtGoal (BioIK)	6.2%	76.7%	93.8%	61.9%	93.6%
Kinematic Decoupling	23.2%	**100.0%**	**100.0%**	**100.0%**	**100.0%**
Straight Seed (KDL)	0.6%	16.6%	56.8%	0.0%	7.0%
Target Seed (KDL)	**48.4%**	60.0%	81.2%	48.2%	53.0%

Table 8.14: Overview of the results on *HydrogenTank*. Light colors are better, the best values are printed in bold. If an approach was tested with multiple solvers, only the best solver is listed.

[25]Timeout of 10s was used

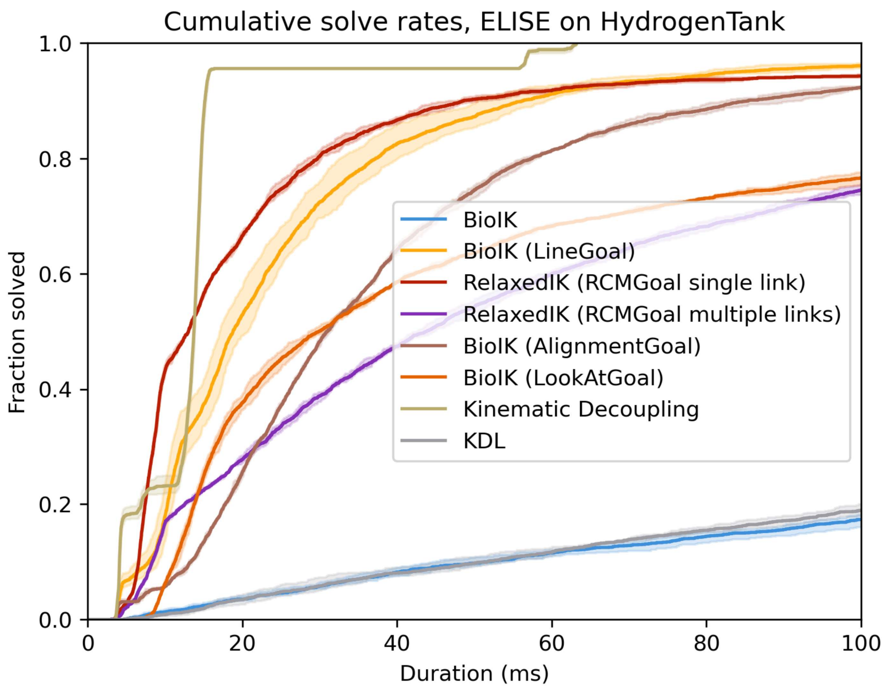


Figure 8.27: Comparison of the best geometry-based goals for *HydrogenTank*

Chapter 9

Conclusion

In this thesis, two research questions were analyzed. The first question was how inverse kinematics solvers could be evaluated through automated methods. To answer this question, the EBIKE library was created and published. The library makes it possible to evaluate different IK solvers that are available as MoveIt plugins. The evaluations can be done on randomly sampled joint values or on objects that provide collision-rich environments. A dataset was developed that makes solver comparisons in various scenarios with different levels of collisions possible. The results of the evaluations on existing solvers highlighted a potential for improvement when collisions are present. Especially endoscopic scenarios, such as hydrogen tanks, proved to be an enormous problem for existing solvers, with as little as 40% solve rate when a timeout of 500ms is used (the default timeout in MoveIt).

The second research question was which approaches could be used to improve these results. For this, the thesis focused on two solvers: BioIK, an evolutionary solver, and RelaxedIK, an optimization-based solver. Since RelaxedIK was only available in Rust, an implementation in C++ usable with MoveIt was developed in this thesis. This implementation was able to outperform all existing solvers available for MoveIt.

Then, several improvements were implemented for both of these solvers. First, rejection-based sampling was implemented, improving the solve rates for the BioIK, TracIK, and PickIK solvers by up to 25.9 percentage points when avoiding self-collisions and by up to 99.7 percentage points avoiding environment collisions. The performance increase that was obtained with this simple improvement highlights that the area of collision checking was neglected by the MoveIt kinematics community. To incorporate collision detection into solvers, two approaches, one based on distance to collisions and one based on penetration depth, were implemented. The evaluations showed that the penetration depth approach was more promising, but both collision operations were too slow to be usable, with more than 98% and 90% of the solver time spent on the distance and depth calculations, respectively. Another approach, based on insights from robot-assisted minimally invasive surgery, uses a remote center of motion, a point through which the robot must always pass before its target is reached. This approach significantly improved the query results for both RelaxedIK and BioIK, reaching more than 90% of the targets for the hydrogen tanks with the small and the large opening within 100ms. Three alternative constraints on dimensions of a link's position ("LineGoal") and orientation ("AlignmentGoal", "LookAtGoal") were proposed. While these approaches also worked well, they were all outperformed by the RCMGoal on one of the tanks. An alternative IK solution strategy of endoscopic robots, kinematic decoupling, was proposed and achieved excellent results on the larger hydrogen tank (100% solve rate within 100ms).

Evaluating the use of MoveIt's IKCostFn to specify custom goals showed that it led to significant overhead on BioIK but not on RelaxedIK. Final analyses of BioIK parameters and RelaxedIK optimization solvers showed that the current parameters were already well-selected and did not allow for large general improvement. The evaluation of seed configurations highlighted the potential of a well-selected seed that can instantly boost solver performance.

The evaluations also determined that the evolutionary solver calls its objective function much more often than the optimization-based solver, making objective function performance optimization more critical. Current collision checking approaches are too slow to be usable in objective functions, but if a speedup of approximately one order of magnitude is reached, usage in RelaxedIK would be possible, while BioIK would require even more speedup. Solutions that add logical constraints, such as a target toward which an end effector should be oriented or a point through which it passes, provide the clearest benefits because they are fast to calculate and avoid most collisions.

9.1 Future Work

As future work, it would be very interesting to include GPU-based collision detection in MoveIt. In the evaluation, it was shown that using penetration depth queries inside a cost function performed well in terms of solver iterations but not in terms of time. Integrating existing GPU-based solutions like [PM12] with 0.002ms per collision check would make these solver techniques feasible.

The next step in incorporating the results into a larger context is to integrate them into a complete planning pipeline. This introduces new challenges such as generating smooth, collision-free trajectories. NVidia's CuRobo framework could provide a solution for that when GPUs are available.

A clean implementation of BioIK 2 would also help to bring forward MoveIt's inverse kinematics solvers. As shown in this thesis, cost functions can improve IK performance in a variety of ways. The PickIK project could be used as a starting point to provide an implementation of BioIK that is more extensible, adaptable, and future-proof. Other solvers could also benefit from the speedup achieved by BioIK 2′s forward kinematics approach.

An approach similar to TracIK is also possible for combining KDL and BioIK. This way, complex constraints and goals can be used, but in situations where they either do not apply or converge slower than KDL, for example on the outside of a container or in other regions of low collisions, the simplicity of KDL is leveraged. Alternatively, the more complex solver could generate a seed to be used by KDL for a range of targets in proximity of this solution. Especially in the area of trajectory planning, where multiple poses that are close to each other have to be covered, this could improve the solve rates.

Parameter tuning remains an important part of IK cost functions. Especially the weights of the goals that were investigated could be tuned to potentially achieve better results. The influence of, e. g., the shape of the groove loss function is also not fully investigated. The foundations for this work have been laid with the EBIKE library.

References

[Aki+19] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A Next-generation Hyperparameter Optimization Framework,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Anchorage AK USA: ACM, Jul. 2019, pp. 2623–2631. doi: 10.1145/3292500.3330701.

[Alm+16] R. Almadhoun, T. Taha, L. Seneviratne, J. Dias, and G. Cai, “A survey on inspecting structures using robotic systems,” *International Journal of Advanced Robotic Systems*, vol. 13, no. 6, p. 172988141666366, Dec. 2016, doi: 10.1177/1729881416663664.

[Arm24] L. Armstrong, “Tesseract.” [Online]. Available: https://github.com/tesseract-robotics/tesseract

[BA15] P. Beeson and B. Ames, “TRAC-IK: An open-source library for improved solving of generic inverse kinematics,” in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, Seoul, South Korea: IEEE, Nov. 2015, pp. 928–935. doi: 10.1109/HUMANOIDS.2015.7363472.

[BHW17] M. Bestmann, N. Hendrich, and F. Wasserfall, “ROS for Humanoid Soccer Robots,” presented at the The 12th Workshop on Humanoid Soccer Robots at 17th IEEE-RAS International Conference on Humanoid Robots, 2017, p. 1–2.

[Ber+11] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, in NIPS'11. Red Hook, NY, USA: Curran Associates Inc., 2011, pp. 2546–2554.

[Bha+21] M. Bhardwaj *et al.*, “STORM: An Integrated Framework for Fast Joint-Space Model-Predictive Control for Reactive Manipulation.,” in *Conference on Robot Learning, 8-11 November 2021, London, UK.*, 2021, pp. 750–759. [Online]. Available: https://proceedings.mlr.press/v164/bhardwaj22a.html

[CB16] E. Coumans and Y. Bai, “PyBullet, a Python module for physics simulation for games, robotics and machine learning.” [Online]. Available: http://pybullet.org/

[CSC24] D. Coleman, I. Sucan, and S. Chitta, “MoveIt KDL Kinematics Plugin.” [Online]. Available: https://github.com/moveit/moveit2/tree/main/moveit_kinematics/kdl_kinematics_plugin

[Cha+22] C. Chamzas *et al.*, “MotionBenchMaker: A Tool to Generate and Benchmark Motion Planning Datasets,” *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 882–889, Apr. 2022, doi: 10.1109/LRA.2021.3133603.

[Che+20] Y. Chen, S. Zhang, Z. Wu, B. Yang, Q. Luo, and K. Xu, “Review of surgical robotic systems for keyhole and endoscopic procedures: state of the art and perspectives,” *Frontiers of Medicine*, vol. 14, no. 4, pp. 382–403, Aug. 2020, doi: 10.1007/s11684-020-0781-x.

[Chi16] S. Chitta, “MoveIt!: An Introduction,” *Robot Operating System (ROS): The Complete Reference (Volume 1).* Springer International Publishing, Cham, pp. 3–27, 2016. doi: 10.1007/978-3-319-26054-9_1.

[Chi97] S. Chiaverini, “Singularity-robust task-priority redundancy resolution for real-time kinematic control of robot manipulators,” *IEEE Transactions on Robotics and Automation*, vol. 13, no. 3, pp. 398–410, Jun. 1997, doi: 10.1109/70.585902.

[Col+23] J. Colan, A. Davila, K. Fozilov, and Y. Hasegawa, “A Concurrent Framework for Constrained Inverse Kinematics of Minimally Invasive Surgical Robots,” *Sensors*, vol. 23, no. 6, p. 3328–3329, Mar. 2023, doi: 10.3390/s23063328.

[Cro22] S. Crozet, “ncollide – 2 and 3-dimensional collision detection library in Rust.” [Online]. Available: https://github.com/dimforge/ncollide

[DCH24] A. Davila, J. Colan, and Y. Hasegawa, “Real-time inverse kinematics for robotic manipulation under remote center-of-motion constraint using memetic evolution,” *Journal of Computational Design and Engineering*, vol. 11, no. 3, pp. 248–264, Jun. 2024, doi: 10.1093/jcde/qwae047.

[Eri05] C. Ericson, *Real-Time Collision Detection.* in The Morgan Kaufmann Series in Interactive 3D Technology. Morgan Kaufmann Publishers, 2005.

[Eul76] L. Euler, “Formulae generales pro translatione quacunque corporum rigidorum,” *Novi Commentarii academiae scientiarum Petropolitanae*, pp. 189–207, 1776.

[Fed01] Federal Aviation Administration, “Transport Airplane Fuel Tank System Design Review, Flammability Reduction, and Maintenance and Inspection Requirements.” [Online]. Available: https://www.govinfo.gov/content/pkg/FR-2001-05-07/pdf/01-10129.pdf

[GC13] E. Galceran and M. Carreras, “A survey on coverage path planning for robotics,” *Robotics and Autonomous Systems*, vol. 61, no. 12, pp. 1258–1276, Dec. 2013, doi: 10.1016/j.robot.2013.09.004.

[GJK88] E. Gilbert, D. Johnson, and S. Keerthi, “A fast procedure for computing the distance between complex objects in three-dimensional space,” *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 193–203, Apr. 1988, doi: 10.1109/56.2083.

[HDW21] F. Heilemann, A. Dadashi, and K. Wicke, “Eeloscope—Towards a Novel Endoscopic System Enabling Digital Aircraft Fuel Tank Maintenance,” *Aerospace*, vol. 8, no. 5, p. 136–137, May 2021, doi: 10.3390/aerospace8050136.

[Ham44] W. R. Hamilton, “On quaternions; or on a new system of imaginaries in Algebra,” *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 25, no. 169, pp. 489–495, 1844.

[KK22] Z. Kingston and L. E. Kavraki, “Robowflex: Robot Motion Planning with MoveIt Made Easy,” in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2022, pp. 3108–3114. doi: 10.1109/IROS47612.2022.9981698.

[Kim+03] Y. J. Kim, M. A. Otaduy, M. C. Lin, and D. Manocha, “Fast penetration depth estimation using rasterization hardware and hierarchical refinement,” in *Proceedings of the nineteenth annual symposium on Computational geometry*, San Diego California USA: ACM, Jun. 2003, pp. 386–387. doi: 10.1145/777792.777856.

[LE93] S. Lie and F. Engel, *Theorie der Transformationsgruppen*, vol. 3. Teubner, 1893.

[MB17] A. McAffee and E. Brynjolfsson, *Machine, Platform, Crowd: Harnessing Our Digital Future.* W. W. Norton & Company, Inc., 2017.

[Mac+22] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, May 2022, doi: 10.1126/scirobotics.abm6074.

[Mon+22] L. Montaut, Q. Le Lidec, V. Petrik, J. Sivic, and J. Carpentier, “Collision Detection Accelerated: An Optimization Perspective,” in *Robotics: Science and Systems XVIII*, Robotics: Science, Systems Foundation, Jun. 2022. doi: 10.15607/RSS.2022.XVIII.039.

[NJ95] T. S. Newman and A. K. Jain, “A Survey of Automated Visual Inspection,” *Computer Vision and Image Understanding*, vol. 61, no. 2, pp. 231–262, Mar. 1995, doi: 10.1006/cviu.1995.1017.

[NWX18] G. Niu, J. Wang, and K. Xu, “Model analysis for a continuum aircraft fuel tank inspection robot based on the Rzeppa universal joint,” *Advances in Mechanical Engineering*, vol. 10, no. 5, p. 168781401877822, May 2018, doi: 10.1177/1687814018778229.

[PCM12] J. Pan, S. Chitta, and D. Manocha, “FCL: A general purpose library for collision and proximity queries,” in *2012 IEEE International Conference on Robotics and Automation*, St Paul, MN, USA: IEEE, May 2012, pp. 3859–3866. doi: 10.1109/ICRA.2012.6225337.

[PM12] J. Pan and D. Manocha, “GPU-based parallel collision detection for fast motion planning,” *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 187–200, Feb. 2012, doi: 10.1177/0278364911429335.

[Pan+15] J. Pan *et al.*, “HPP-FCL: an extension of the Flexible Collision Library.” [Online]. Available: https://github.com/humanoid-path-planner/hpp-fcl

[RM23] O. Rippel and D. Merhof, “Anomaly Detection for Automated Visual Inspection: A Review,” *Bildverarbeitung in der Automation*, vol. 17. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–13, 2023. doi: 10.1007/978-3-662-66769-9_1.

[RMG18] D. Rakita, B. Mutlu, and M. Gleicher, “RelaxedIK: Real-time Synthesis of Accurate and Feasible Robot Arm Motion,” in *Robotics: Science and Sys-*

tems XIV, Robotics: Science, Systems Foundation, Jun. 2018. doi: 10.15607/RSS.2018.XIV.043.

[Rae23] M. Raessa, “MoveIt Inverse Kinematics Benchmarking.” [Online]. Available: https://github.com/PickNikRobotics/ik_benchmarking

[Rak+21] D. Rakita, H. Shi, B. Mutlu, and M. Gleicher, “CollisionIK: A Per-Instant Pose Optimization Method for Generating Robot Motions with Environment Collision Avoidance,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, Xi'an, China: IEEE, May 2021, pp. 9995–10001. doi: 10.1109/ICRA48506.2021.9561505.

[Rak24] D. Rakita, “Relaxed IK ROS 2.” [Online]. Available: https://github.com/uwgraphics/relaxed_ik_ros2

[Ric24] A. Ricardez Ortigosa, “FuTaMa2 Project.” [Online]. Available: https://github.com/DLR-MO/FuTaMa2

[Rip23] M. Rippberger, “Robotic Evaluation And Comparison Heuristic.” [Online]. Available: https://github.com/ros-industrial/reach

[Rup+18] P. Ruppel, N. Hendrich, S. Starke, and J. Zhang, “Cost Functions to Specify Full-Body Motion and Multi-Goal Manipulation Tasks,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, QLD: IEEE, May 2018, pp. 3152–3159. doi: 10.1109/ICRA.2018.8460799.

[Rup17] P. S. Ruppel, “Performance optimization and implementation of evolutionary inverse kinematics in ROS,” 2017.

[SAO21] R. Smits, E. Aertbelien, and Orocos Developers, “Orocos Kinematics and Dynamics C++ library.” [Online]. Available: https://www.orocos.org/kdl.html

[SHZ19] S. Starke, N. Hendrich, and J. Zhang, “Memetic Evolution for Generic Full-Body Inverse Kinematics in Robotics and Animation,” *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 3, pp. 406–420, Jun. 2019, doi: 10.1109/TEVC.2018.2867601.

[SK16] B. Siciliano and O. Khatib, Eds., “Springer Handbook of Robotics.” in Springer Handbooks. Springer International Publishing, Cham, 2016. doi: 10.1007/978-3-319-32552-1.

[Sel05] J. M. Selig, “Lie Groups and Lie Algebras in Robotics,” *Computational Noncommutative Algebra and Applications*, vol. 136. Kluwer Academic Publishers, Dordrecht, pp. 101–125, 2005. doi: 10.1007/1-4020-2307-3_5.

[Sta+16] S. Starke, N. Hendrich, S. Magg, and J. Zhang, “An efficient hybridization of Genetic Algorithms and Particle Swarm Optimization for inverse kinematics,” in *2016 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Qingdao, China: IEEE, Dec. 2016, pp. 1782–1789. doi: 10.1109/ROBIO.2016.7866587.

[Sun+23] B. Sundaralingam *et al.*, “CuRobo: Parallelized Collision-Free Robot Motion Generation,” in *2023 IEEE International Conference on Robotics and Au-*

tomation (ICRA), London, United Kingdom: IEEE, May 2023, pp. 8112–8119. doi: 10.1109/ICRA48891.2023.10160765.

[Sun+23] B. Sundaralingam *et al.*, “cuRobo: Parallelized Collision-Free Minimum-Jerk Robot Motion Generation, Revised Technical Report,” Nov. 2023. Accessed: Aug. 09, 2024. [Online]. Available: http://arxiv.org/abs/2310.17274

[Van01] G. Van Den Bergen, “Proximity queries and penetration depth computation on 3d game objects,” presented at the Game developers conference, 2001, p. 209–210.

[WAT15] G. Williams, A. Aldrich, and E. Theodorou, “Model Predictive Path Integral Control using Covariance Variable Importance Sampling.” Accessed: Aug. 13, 2024. [Online]. Available: http://arxiv.org/abs/1509.01149

[Wan+23] Y. Wang, P. Praveena, D. Rakita, and M. Gleicher, “RangedIK: An Optimization-based Robot Motion Generation Method for Ranged-Goal Tasks,” in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, London, United Kingdom: IEEE, May 2023, pp. 9700–9706. doi: 10.1109/ICRA48891.2023.10161311.

[Wea24] T. Weaver, “pick_ik: Inverse Kinematics solver for MoveIt.” [Online]. Available: https://github.com/PickNikRobotics/pick_ik

Appendix A

Additional Results

This chapter contains some additional results that were not included into the main part of the thesis either because they did not deviate from the selected results or for keeping the thesis more structured.

A1 Baseline

The baseline results for *Barrel*, *SmallTable*, and *TableObjects* were not shown. They match the presented results where KDL performs best, BioIK and TracIK quickly stop solving and PickIK has a slow performance.

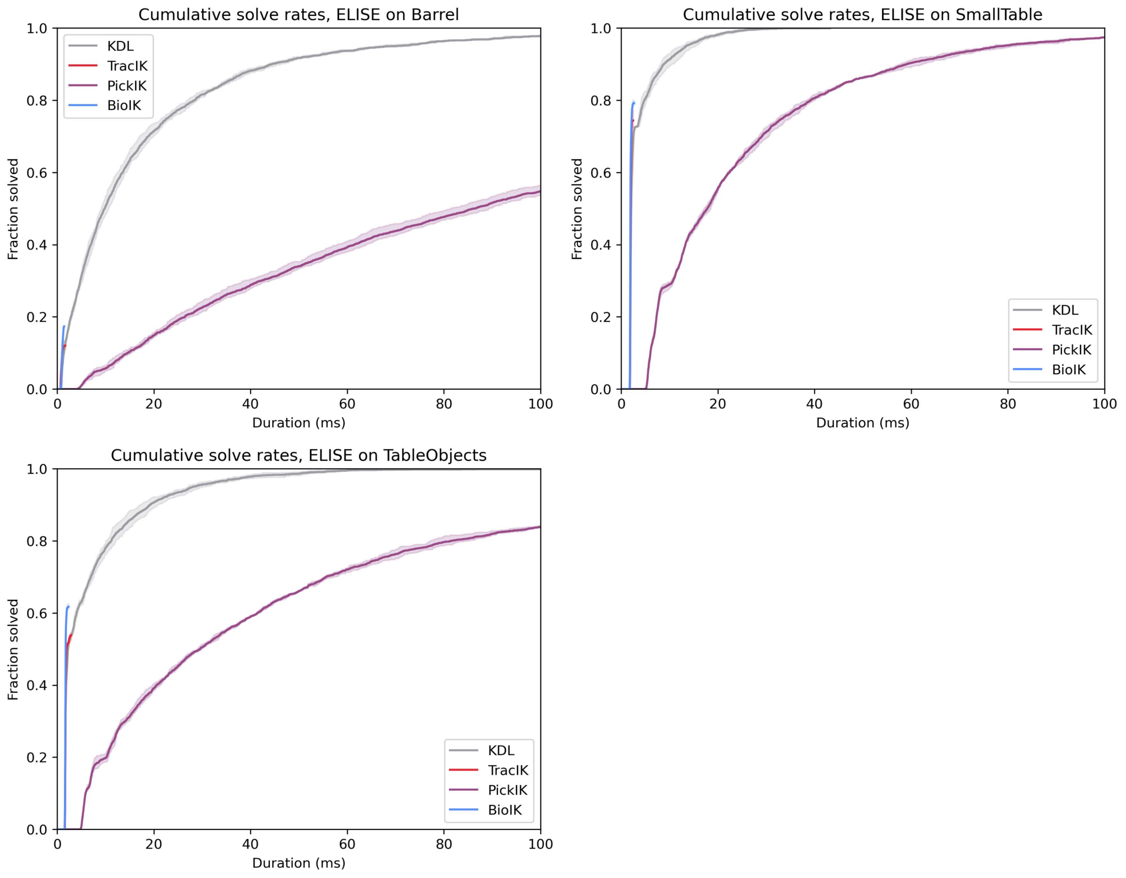


Figure A1: Cumulative solve rates for other scenarios, view cropped to 100ms timeout

A2 RelaxedIK Baseline

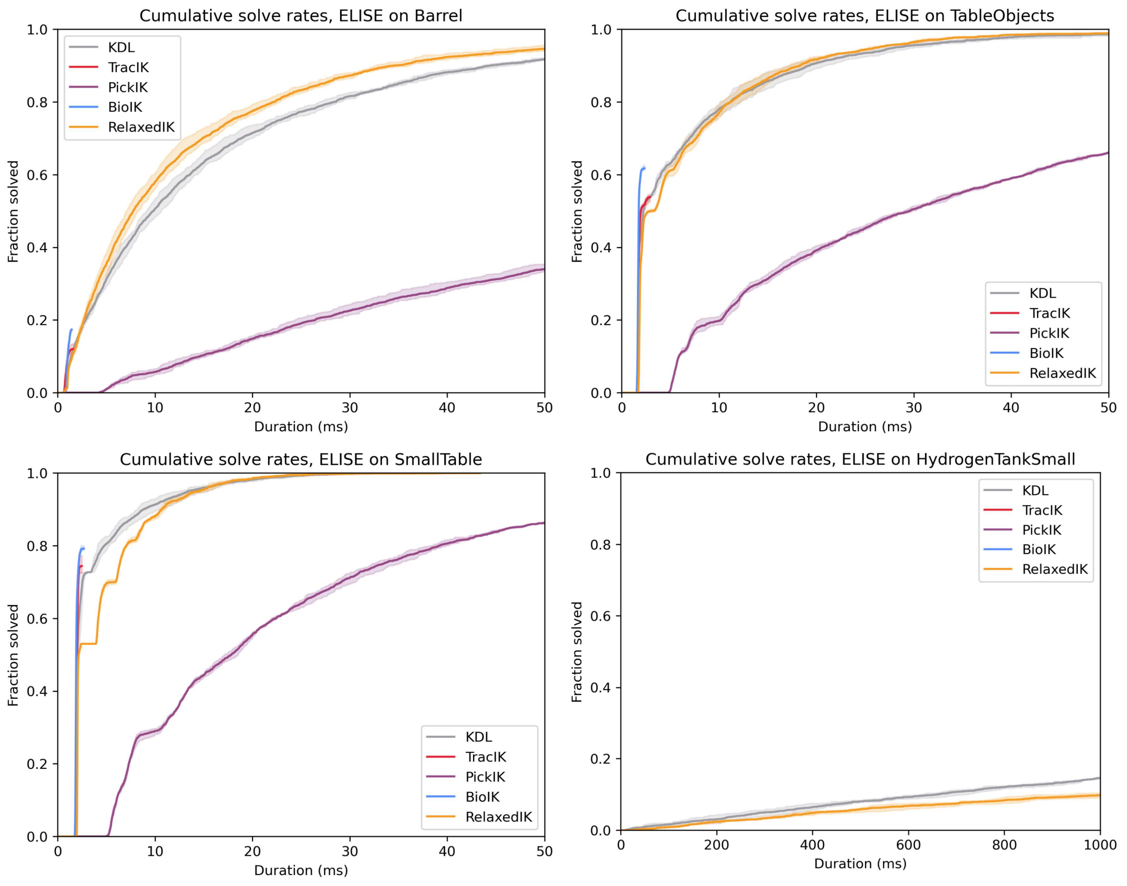


Figure A2: Comparison of RelaxedIK to existing solvers on additional scenarios. It is better than KDL on *Barrel*, has a similar performance on *TableObjects* and *SmallTable*, and performs slightly worse on *HydrogenTankSmall*.

A3 Quantitative Baseline Results

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	31.1%	50.5%	97.8%	99.9%	19.966ms
TracIK	12.1%	12.1%	12.1%	12.1%	0.915ms
PickIK	0.9%	5.7%	54.7%	63.3%	53.356ms
BioIK	17.4%	17.4%	17.4%	17.4%	1.039ms
RelaxedIK	35.3%	57.9%	98.4%	99.8%	16.149ms

Iterations	1	2	3	5	10	100	Mean
KDL	11.6%	18.0%	23.2%	34.5%	52.4%	97.6%	19.753
TracIK	12.1%	12.1%	12.1%	12.1%	12.1%	12.1%	1.000
PickIK	8.2%	15.5%	22.1%	32.9%	50.5%	63.3%	6.340
BioIK	17.4%	17.4%	17.4%	17.4%	17.4%	17.4%	1.000
RelaxedIK	9.6%	18.1%	25.5%	36.8%	57.1%	98.1%	17.487

Table A1: Baseline on *Barrel*

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	63.0%	78.0%	99.9%	100.0%	7.448ms
TracIK	53.9%	53.9%	53.9%	53.9%	1.776ms
PickIK	0.7%	19.8%	83.9%	87.8%	33.997ms
BioIK	61.8%	61.8%	61.8%	61.8%	1.719ms
RelaxedIK	61.1%	76.9%	99.8%	100.0%	7.582ms

Iterations	1	2	3	5	10	100	Mean
KDL	54.7%	64.4%	71.6%	81.8%	92.6%	100.0%	3.476
TracIK	53.9%	53.9%	53.9%	53.9%	53.9%	53.9%	1.000
PickIK	26.3%	41.9%	52.0%	65.8%	79.9%	87.8%	4.144
BioIK	61.8%	61.8%	61.8%	61.8%	61.8%	61.8%	1.000
RelaxedIK	50.1%	61.5%	69.1%	80.4%	92.6%	100.0%	3.720

Table A2: Baseline on *Shelf*

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	6.6%	22.1%	87.4%	100.0%	46.987ms
TracIK	0.7%	0.7%	0.7%	0.7%	2.773ms
PickIK	0.0%	3.8%	40.7%	53.9%	65.505ms
BioIK	0.3%	0.3%	0.3%	0.3%	2.821ms
RelaxedIK	10.9%	25.1%	91.3%	100.0%	39.486ms

Iterations	1	2	3	5	10	100	Mean
KDL	0.2%	9.6%	18.4%	31.9%	52.7%	98.7%	17.252
TracIK	0.7%	0.7%	0.7%	0.7%	0.7%	0.7%	1.000
PickIK	8.3%	15.4%	22.0%	31.9%	47.7%	53.9%	5.325
BioIK	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%	1.000
RelaxedIK	6.0%	15.4%	23.3%	35.7%	57.1%	99.3%	14.782

Table A3: Baseline on *Table*

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	81.9%	89.3%	100.0%	100.0%	4.544ms
TracIK	80.5%	80.5%	80.5%	80.5%	1.854ms
PickIK	0.0%	26.3%	80.3%	95.6%	53.567ms
BioIK	85.1%	85.1%	85.1%	85.1%	1.970ms
RelaxedIK	92.0%	96.2%	100.0%	100.0%	2.877ms

Iterations	1	2	3	5	10	100	Mean
KDL	76.6%	82.1%	86.0%	91.3%	96.7%	100.0%	2.178
TracIK	80.5%	80.5%	80.5%	80.5%	80.5%	80.5%	1.000
PickIK	45.7%	65.1%	76.0%	87.7%	95.3%	95.6%	2.347
BioIK	85.1%	85.1%	85.1%	85.1%	85.1%	85.1%	1.000
RelaxedIK	89.1%	91.8%	93.9%	96.4%	98.8%	100.0%	1.465

Table A4: Baseline on *SmallTable*

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	80.4%	91.3%	100.0%	100.0%	3.963ms
TracIK	74.4%	74.4%	74.4%	74.4%	1.966ms
PickIK	0.0%	29.0%	97.4%	99.2%	25.411ms
BioIK	79.2%	79.2%	79.2%	79.2%	1.928ms
RelaxedIK	69.5%	88.0%	100.0%	100.0%	4.919ms

Iterations	1	2	3	5	10	100	Mean
KDL	72.7%	82.2%	88.4%	94.6%	99.6%	100.0%	1.790
TracIK	74.4%	74.4%	74.4%	74.4%	74.4%	74.4%	1.000
PickIK	40.5%	62.7%	76.5%	88.8%	97.0%	99.2%	2.726
BioIK	79.2%	79.2%	79.2%	79.2%	79.2%	79.2%	1.000
RelaxedIK	53.0%	70.0%	81.6%	92.6%	99.4%	100.0%	2.271

Table A5: Baseline on *TableObjects*

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	0.1%	0.9%	18.9%	56.8%	290.570ms
TracIK	0.0%	0.0%	0.0%	0.0%	0.000ms
PickIK	0.0%	0.0%	0.4%	0.5%	72.124ms
BioIK	0.0%	0.0%	0.0%	0.0%	0.788ms
RelaxedIK	0.6%	1.1%	14.1%	48.4%	311.911ms

Iterations	1	2	3	5	10	100	Mean
KDL	0.0%	0.7%	1.2%	3.2%	7.2%	37.4%	86.743
TracIK	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000
PickIK	0.0%	0.1%	0.2%	0.3%	0.5%	0.5%	5.722
BioIK	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.333
RelaxedIK	0.6%	1.0%	1.6%	2.6%	5.1%	29.3%	100.581

Table A6: Baseline on *HydrogenTank*

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	0.0%	0.0%	1.6%	14.5%	476.454ms
TracIK	0.0%	0.0%	0.0%	0.0%	0.000ms
PickIK	0.0%	0.0%	0.1%	0.1%	15.055ms
BioIK	0.0%	0.0%	0.0%	0.0%	0.000ms
RelaxedIK	0.0%	0.0%	0.8%	9.8%	451.231ms

Iterations	1	2	3	5	10	100	Mean
KDL	0.0%	0.1%	0.2%	0.5%	1.0%	7.5%	101.495
TracIK	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000
PickIK	0.0%	0.0%	0.0%	0.1%	0.1%	0.1%	1.167
BioIK	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.000
RelaxedIK	0.0%	0.0%	0.0%	0.1%	0.4%	5.3%	99.984

Table A7: Baseline on *HydrogenTankSmall*

A4 Rejection Sampling

The results after adding rejection sampling are shown here for *HydrogenTank* and *HydrogenTankSmall* to give context for the tables in the following sections.

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	0.1%	0.9%	18.9%	56.8%	290.570ms
BioIK	0.1%	1.3%	17.3%	68.9%	328.166ms
RelaxedIK	0.6%	1.1%	14.1%	48.4%	311.911ms

Iterations	1	2	3	10	Mean
KDL	0.0%	0.7%	1.2%	7.2%	86.743 its
BioIK	0.0%	1.1%	1.7%	6.4%	110.758 its
RelaxedIK	0.6%	1.0%	1.6%	5.1%	100.581 its

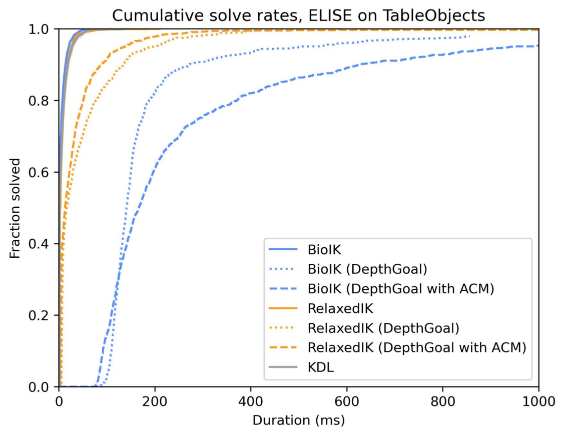
Table A8: Base results on *HydrogenTank*, all solvers use rejection sampling

Duration	5ms	10ms	100ms	1000ms	Mean
KDL	0.0%	0.0%	1.6%	14.5%	476.454ms
BioIK	0.0%	0.0%	1.4%	13.8%	472.623ms
RelaxedIK	0.0%	0.0%	0.8%	9.8%	451.231ms

Iterations	1	2	3	10	Mean
KDL	0.0%	0.1%	0.2%	1.0%	101.495 its
BioIK	0.0%	0.1%	0.2%	0.4%	104.775 its
RelaxedIK	0.0%	0.0%	0.0%	0.4%	99.984 its

Table A9: Base results on *HydrogenTankSmall*, all solvers use rejection sampling

A5 Penetration Depth



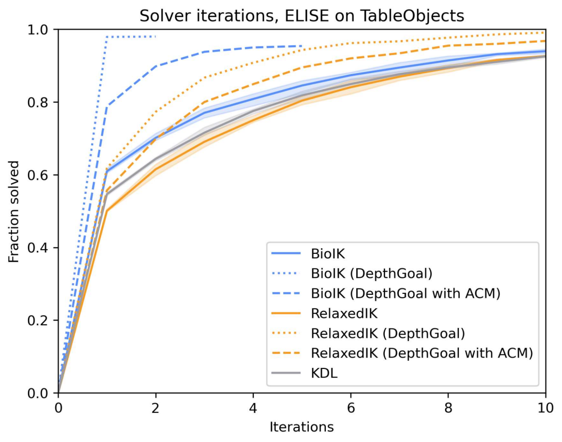


Figure A3: Results of penetration depth goal on *TableObjects*. As for the fuel tank scenarios, the solvers perform worse in terms of time but better in terms of solver iterations.

A6 Trials with High Timeout

Long runs (10s timeout) for collision distance and penetration depth were also performed for the *Shelf* and *TableObjects* scenarios.

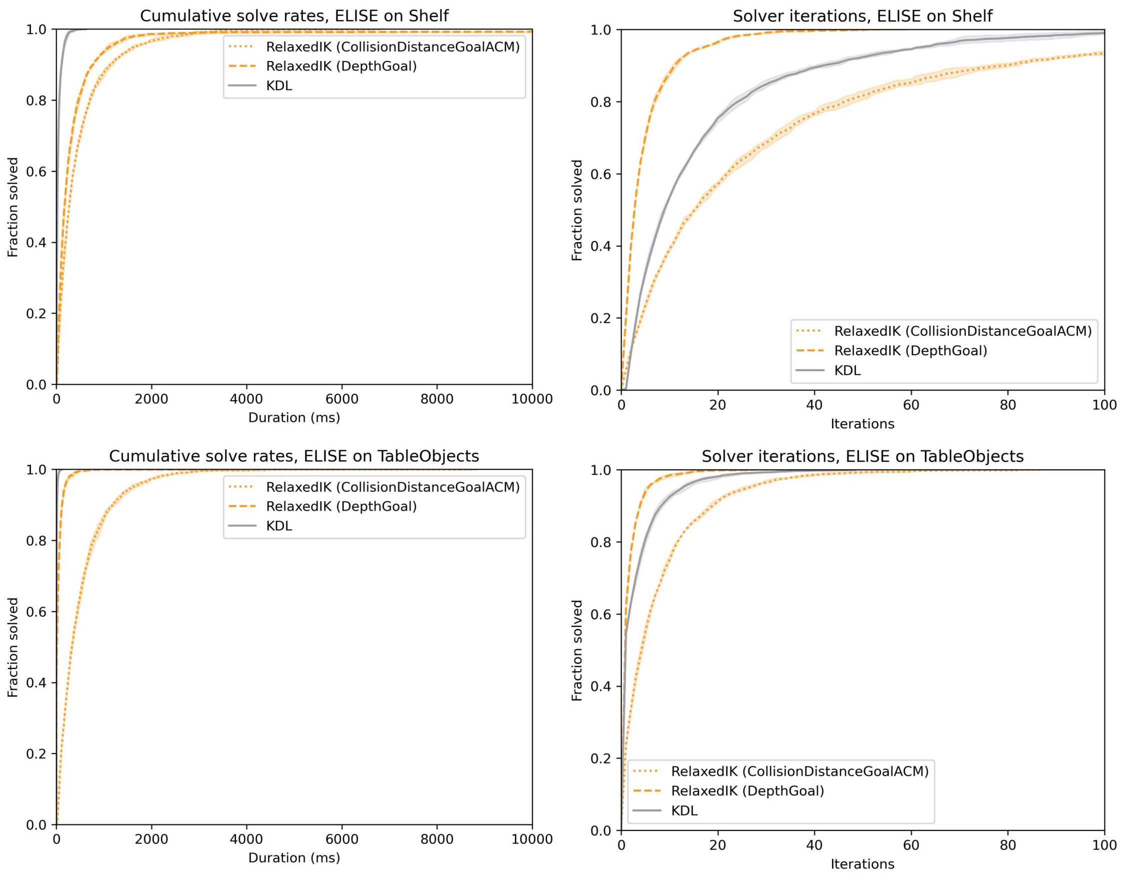
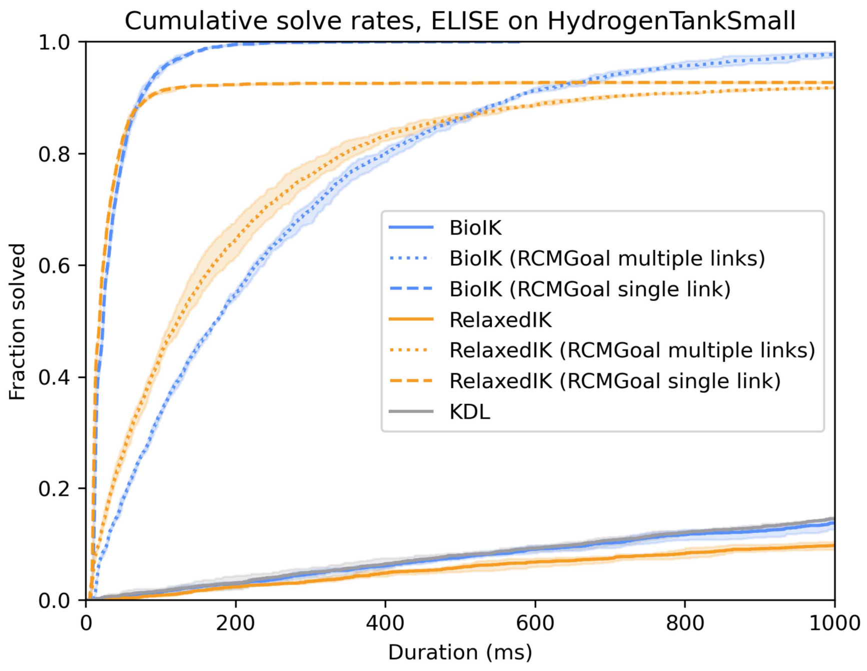


Figure A4: Results of trials with 10s timeout for penetration depth and collision distance approaches on *Shelf* and *TableObjects*. As for *HydrogenTank* and *HydrogenTankSmall*, the performance of the penetration depth objective is good in terms of solver iterations, the performance of the collision distance objective is bad.

A7 RCM Objective



Duration	5ms	10ms	100ms	1000ms
BioIK	0.0%	0.0%	1.4%	13.8%
BioIK (RCMGoal multiple links)	0.0%	0.0%	33.4%	97.7%
BioIK (RCMGoal single link)	0.0%	7.2%	**95.0%**	**100.0%**
RelaxedIK	0.0%	0.0%	0.8%	9.8%
RelaxedIK (RCMGoal multiple links)	0.0%	4.5%	43.8%	91.8%
RelaxedIK (RCMGoal single link)	0.0%	**16.9%**	91.1%	92.7%

Figure A5: Comparison of different RCMGoals on *HydrogenTankSmall*

A8 Collision Point Distance

Duration	5ms	10ms	100ms	1000ms	Mean
HydrogenTank	3.0%	34.7%	94.5%	97.5%	31.485ms
HydrogenTankSmall	0.0%	3.9%	68.3%	89.6%	79.082ms

Iterations	1	2	3	10	Mean
HydrogenTank	54.9%	73.5%	83.7%	97.3%	2.054 its
HydrogenTankSmall	15.3%	27.3%	38.0%	73.7%	6.369 its

Table A10: Results of collision point distance (RelaxedIK) on *HydrogenTank* and *HydrogenTankSmall*

A9 Line Objective

Duration	5ms	10ms	100ms	1000ms	Mean
BioIK (LineGoal)	6.7%	17.9%	96.1%	97.9%	26.66ms
RelaxedIK (LineGoal)	2.2%	19.2%	95.3%	97.3%	31.93ms

Iterations	1	2	3	10	Mean
BioIK (LineGoal)	27.0%	46.9%	62.1%	92.6%	4.015 its
RelaxedIK (LineGoal)	37.9%	50.0%	60.6%	90.2%	3.959 its

Table A11: Results of the line objective on *HydrogenTank*

Duration	5ms	10ms	100ms	1000ms	Mean
BioIK (LineGoal)	0.0%	5.3%	71.8%	79.1%	45.26ms
RelaxedIK (LineGoal)	0.0%	2.8%	56.7%	80.2%	98.46ms

Iterations	1	2	3	10	Mean
BioIK (LineGoal)	18.0%	33.8%	43.2%	70.9%	5.075 its
RelaxedIK (LineGoal)	18.5%	28.6%	36.7%	64.6%	7.330 its

Table A12: Results of the line objective on *HydrogenTankSmall*

A10 Alignment Objective

Duration	5ms	10ms	100ms	1000ms	Mean
BioIK (AlignmentGoal)	3.1%	5.7%	92.4%	96.7%	39.79ms
RelaxedIK (AlignmentGoal)	5.4%	8.1%	81.5%	96.6%	57.95ms

Iterations	1	2	3	10	Mean
BioIK (AlignmentGoal)	10.9%	31.6%	46.6%	87.8%	5.114 its
RelaxedIK (AlignmentGoal)	24.9%	32.9%	40.6%	73.4%	7.596 its

Table A13: Results on the alignment objective on *HydrogenTank*

Duration	5ms	10ms	100ms	1000ms	Mean
BioIK (AlignmentGoal)	0.0%	0.0%	0.0%	0.0%	–
RelaxedIK (AlignmentGoal)	0.0%	0.0%	0.3%	1.4%	515.6ms

Iterations	1	2	3	10	Mean
BioIK (AlignmentGoal)	0.0%	0.0%	0.0%	0.0%	–
RelaxedIK (AlignmentGoal)	0.0%	0.0%	0.0%	0.3%	53.92 its

Table A14: Results on the alignment objective on *HydrogenTankSmall*

A11 Look At Objective

Duration	5ms	10ms	100ms	1000ms	Mean
HydrogenTank	0.0%	6.2%	76.7%	93.8%	79.60ms
HydrogenTankSmall	0.0%	0.9%	79.2%	96.5%	77.59ms

Iterations	1	2	3	10	Mean
HydrogenTank	61.9%	79.0%	86.8%	93.6%	1.668 its
HydrogenTankSmall	64.9%	82.5%	89.4%	96.3%	1.636 its

Table A15: Quantitative results for look at objective (BioIK) on *HydrogenTank* and *HydrogenTankSmall*

A12 Kinematic Decoupling

Using kinematic decoupling, only a single call to the solution callback (solver iteration) is used. Therefore, the final solve rate is reached relatively quick (80ms for *HydrogenTank*, 20ms for *HydrogenTankSmall*). The table shows the results after 5ms, 10ms, and 100ms for better comparability with other solvers.

Duration	5ms	10ms	100ms
HydrogenTank	18.1%	23.2%	100.0%
HydrogenTankSmall	0.0%	6.6%	54.0%

Table A16: Results of kinematic decoupling, only a single call to the solution callback is used.

A13 RelaxedIK Cost Functions

On RelaxedIK, the overhead of using a MoveIt IKCostFn is negligible.

Duration	5ms	10ms	100ms	1000ms	Mean
RelaxedIK	100.0%	100.0%	100.0%	100.0%	0.436ms
RelaxedIK (Empty CostFn)	99.9%	100.0%	100.0%	100.0%	0.448ms

Solver	1	2	3	10	Mean
RelaxedIK	86.0%	95.4%	98.2%	100.0%	1.216 its
RelaxedIK (Empty CostFn)	86.5%	95.3%	97.9%	99.9%	1.235 its

Table A17: Results of RelaxedIK with and without an IKCostFn on *Random*

A14 Surface Inspection

The results for the ScanGoal on *HydrogenTankSmall* show that the ScanGoal performs better than on *HydrogenTank*, independent of whether is is combined with the RCMGoal or not.

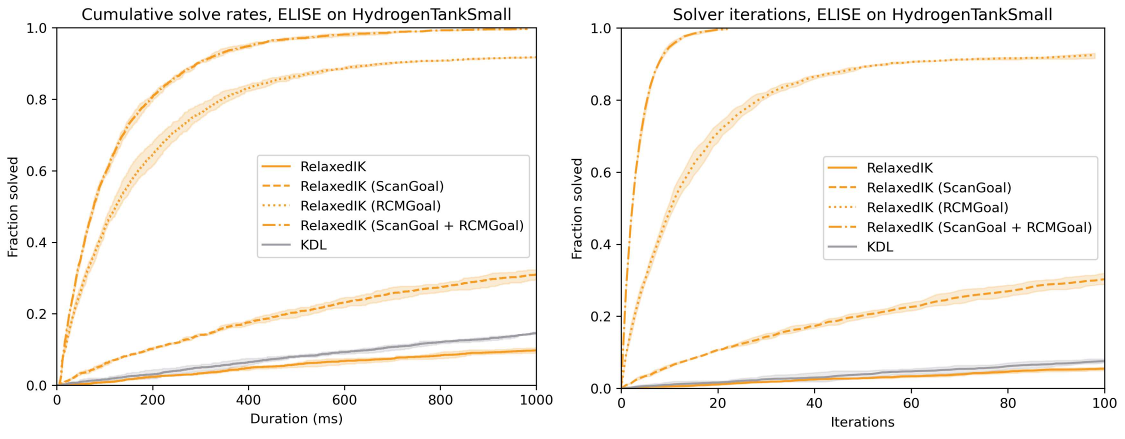


Figure A6: ScanGoal on *HydrogenTankSmall*

A15 Final Results on HydrogenTankSmall

Approach	**Timeout**			**Iterations**	
	10ms	**100ms**	**1000ms**	**1**	**10**
Baseline (KDL)	0.0%	1.6%	14.5%	0.0%	1.0%
RelaxedIK	0.0%	0.8%	9.8%	0.0%	0.4%
Rejection Sampling (BioIK)	0.0%	1.4%	13.8%	0.0%	0.4%
Collision Distance (RelaxedIK)	0.0%	0.0%	0.0%	0.0%[26]	0.5%[26]
Penetration Depth (RelaxedIK)	0.0%	0.4%	3.3%	0.2%[26]	2.5%[26]
RCMGoal single link (RelaxedIK)	**16.9%**	**91.1%**	92.7%	49.5%	92.3%
RCMGoal multiple links (RelaxedIK)	4.5%	43.8%	91.8%	9.3%	49.1%
Collision point distance (RelaxedIK)	3.9%	68.3%	89.6%	15.3%	73.7%
LineGoal (BioIK)	5.3%	71.8%	79.1%	18.0%	70.9%
AlignmentGoal (BioIK)	0.0%	0.0%	0.0%	0.0%	0.0%
LookAtGoal (BioIK)	0.9%	79.2%	**96.5%**	**64.9%**	**96.3%**
Kinematic Decoupling	0.0%	6.6%	54.0%	54.0%	54.0%
Straight Seed (KDL)	0.0%	1.9%	16.1%	0.0%	0.9%
Target Seed (KDL)	26.3%	27.6%	39.1%	26.4%	27.0%

Table A18: Overview of the results on *HydrogenTankSmall*. Light colors are better, the best values are printed in bold. If an approach was tested with multiple solvers, only the best solver is listed.

[26]Timeout of 10s was used

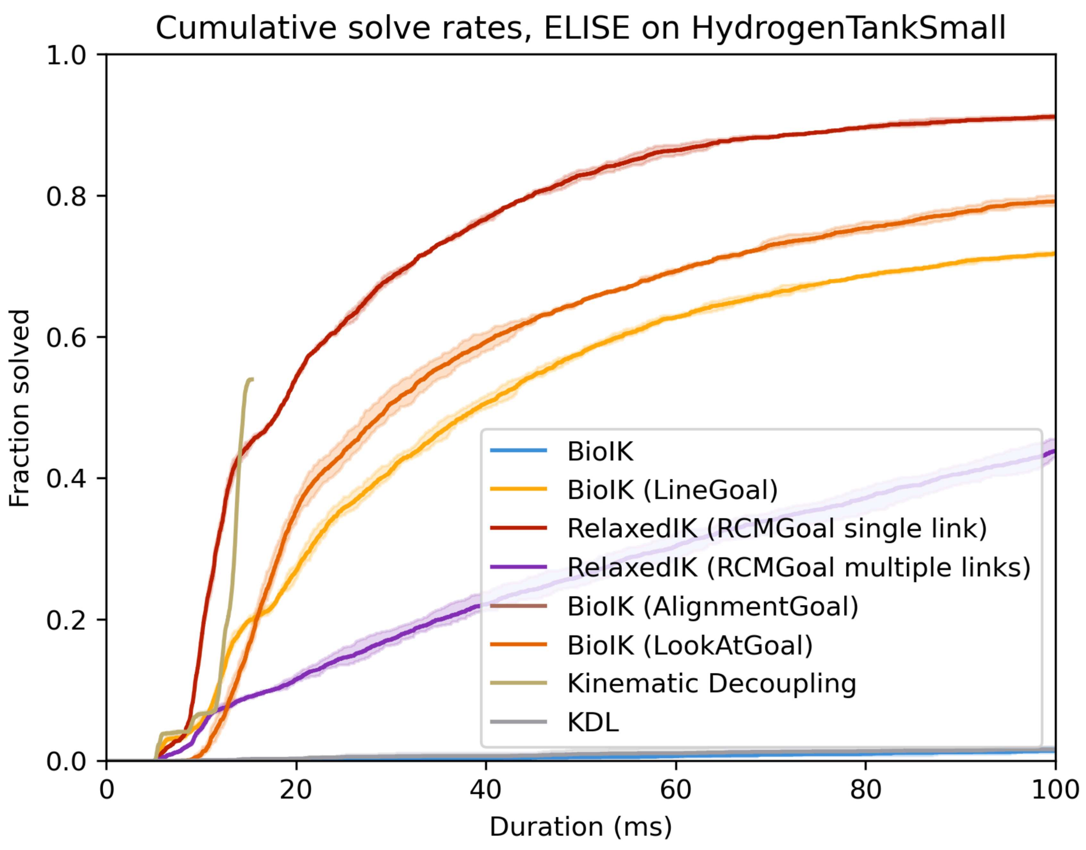


Figure A7: Comparison of the best geometry-based goals for *HydrogenTankSmall*

Appendix B

Open Source Contributions

This table lists all repositories, pull requests, and issues opened as part of the work on this thesis.

Project	**Description**	**Link**
<> Repositories		
EBIKE	Enhancement and Benchmarking of Inverse Kinematics in Environments	https://github.com/DLR-MO/ebike
RelaxedIK C++	C++ implementation of RelaxedIK for ROS 2 and MoveIt	https://github.com/timonegk/relaxed_ik_cpp
Collision Benchmarking	REACH plugin, EBIKE scenarios, and IK configurations for collision benchmarking	https://github.com/timonegk/collision_benchmarking
⊙ Issues		
PickIK	Highlight differences to BioIK	https://github.com/PickNikRobotics/pick_ik/issues/67#issuecomment-2283706482
PickIK	Wipeout should not remove the best individual	https://github.com/PickNikRobotics/pick_ik/issues/72
REACH	Segmentation fault when using a plugin	https://github.com/ros-industrial/reach/issues/70
MoveIt	Make collision environment available to kinematics solvers	https://github.com/moveit/moveit2/issues/2856

Pull Requests		
TracIK	Add rejection sampling	https://bitbucket.org/traclabs/trac_ik/pull-requests/38
BioIK	Add rejection sampling	https://github.com/PickNikRobotics/bio_ik/pull/21
BioIK	Fix cppoptlib solvers	https://github.com/PickNikRobotics/bio_ik/pull/22
BioIK	Make parameters configurable	https://github.com/PickNikRobotics/bio_ik/pull/23
PickIK	Fix rejection sampling	https://github.com/PickNikRobotics/pick_ik/pull/73
REACH	Keep plugin factories in scope	https://github.com/ros-industrial/reach/pull/74
REACH	Replace deprecated usages of boost	https://github.com/ros-industrial/reach/pull/72
REACH	Add ik_time to ReachRecord	https://github.com/ros-industrial/reach/pull/75
REACH	Fix CMake error about missing target MPI::MPI_C	https://github.com/ros-industrial/reach/pull/82
REACH ROS 2	Fix deprecated header for tf2_eigen	https://github.com/ros-industrial/reach_ros2/pull/35
REACH ROS 2	Change durability policy of mesh display to transient_local	https://github.com/ros-industrial/reach_ros2/pull/36