Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Performance optimization and implementation of evolutionary inverse kinematics in ROS

Masterthesis
at Research Group Technical Aspects of Multimodal Systems, TAMS
Prof. Dr. Jianwei Zhang

Department of Informatics
MIN-Faculty
Universität Hamburg

submitted by
Philipp Sebastian Ruppel
Course of study: Informatik
Matrikelnr.: 6248024
on
22.6.2017

Examiners: Prof. Dr. Jianwei Zhang
Dr. Norman Hendrich

Abstract

In this work, a memetic inverse kinematics solver is developed for the motion planning framework MoveIt and the robot operating system ROS. Inverse kinematics solvers that are already available for MoveIt are limited to kinematic chains with a single end effector. The newly developed solver supports kinematic trees with multiple end effectors. The memetic algorithm uses a combination of evolutionary optimization, particle swarm optimization, and gradient based methods, allowing efficient handling of local minima and joint limits as well as fast convergence to accurate solutions.

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein auf memetischer Optimierung basierender Inverskinematik-Löser for die Bewegungsplanungsbibliothek MoveIt und das Roboterbetriebssystem ROS entwickelt. Während für MoveIt bereits verfügbare IK-Löser auf kinematische Ketten mit jeweils nur einem Endeffektor beschränkt sind, unterstützt der neu entwickelte IK-Löser kinematische Bäume mit mehreren Endeffektoren. Der memetische Algorithmus kombiniert evolutionäre Optimierung mit Partikelschwarmoptimierung und gradientenbasierten Verfahren, wodurch schnelle Konvergenz auf genaue Lösungen bei gleichzeitig hoher Robustheit gegenüber lokalen Minima und Gelenkwinkelgrenzen erreicht wird.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In robotics, inverse kinematics is used to find joint parameters (e.g. joint angles), for which an end effector reaches a certain position and/or orientation (single-goal IK), or for which multiple end effectors reach multiple goals (multi-goal IK). Inverse kinematics is also used for other applications, e.g. character animation and game development.

For simple cases, equations for directly computing inverse kinematics solutions can be found analytically or geometrically. However, this is not always practical, and specialized solvers designed this way are limited to a specific robot or to a small set of supported robots.

A more general approach computes inverse kinematics solutions numerically. Candidate solutions are iteratively improved, until an error function is small enough. Numerical inverse kinematics solvers often use gradient based optimization methods.

However, inverse kinematics problems can exhibit local minima, around which the gradients point towards a suboptimal solution at the local minimum. A small movement away from the local minimum would at first always increase the error. Biologically inspired optimization methods exist, which are typically less susceptible to local minima (e.g. evolutionary optimization and particle swarm optimization). On the other hand, if local minima are not a problem, gradient based methods typically converge faster to accurate solutions. Memetic algorithms combine evolutionary optimization with other methods (e.g. gradient based ones) and can achieve fast convergence to accurate solutions as well as robustness against local minima.

MoveIt is a motion planning framework for the robot operating system ROS. Inverse kinematics solvers currently available in MoveIt are limited to single-goal IK.

Recently, an evolutionary multi-goal inverse kinematics solver has been developed and implemented in C# for the Unity3D game engine at the TAMS research group.

In this work, the evolutionary algorithm is first ported to C++ and integrated into MoveIt as an inverse kinematics plugin. Next, a few minor optimizations and algorithmic changes are made to increase the performance without changing the

general structure of the algorithm. For comparison, several other approaches are implemented as well, including gradient based optimization methods. Finally, a re-designed memetic inverse kinematics solver is developed according to robotics-specific requirements. Different approaches are compared through benchmarks for single-goal as well as multi-goal IK problems. The re-designed memetic IK solver outperforms other methods, including gradient based methods which have been implemented for comparison, as well as already existing gradient-based single-goal IK solvers which were already available for MoveIt.

Chapter 2

State of the Art

2.1 Robot Operating System (ROS)

ROS [1] [2] is a software framework for developing robot software. ROS supports modularization and code reuse through packages, nodes, libraries, and messages.

2.1.1 Packages

A package can contain ROS nodes, library code, and data and configuration files. Information about a package and dependencies between packages are described through manifest files. ROS packages are typically built using ROS-specific build systems (such as catkin, based on CMake).

2.1.2 Nodes

ROS nodes are executable units. A ROS-based system typically consists of multiple nodes running in parallel and communicating with each other through ROS messages. A ROS node can also load and use libraries which are provided by other packages.

2.1.3 Topics and Messages

Messages provide a way for nodes to communicate with each other. Messages are organized via topics. A node can publish messages on a specific topic. Other nodes subscribed to the same topic can receive the published messages. Messages are automatically serialized and de-serialized. The data structures are defined via message description files and are translated to different programming languages during compilation.

2.1.4 Services

ROS services implement procedure call semantics on top of ROS messages. A server can advertise a service. When a client calls the service, it sends a request message

to the server, and when the server has handled the request, it sends back a reply message.

2.1.5 Libraries

ROS packages can expose program libraries to other packages. A ROS library written in C++ can be used by declaring the dependency in the package manifest and including the library header. The build system links the node's executable to the library. ROS libraries can also be used as plug-ins. One package defines a common base class and exposes it through a C++ header. Other packages include the header, provide implementations, and declare the implementations via their package manifests and via ROS plug-in description files. A ROS node can now dynamically enumerate, load and use the plug-ins.

2.1.6 Package Repository

A large number of existing ROS packages is available at the ROS package repositories. [3]

2.1.7 MoveIt

MoveIt [4] [5] is a motion planning framework for ROS. MoveIt supports forward and inverse kinematics, collision checking, perception, and trajectory planning. Inverse kinematics solvers and motion planners can be implemented as plug-ins.

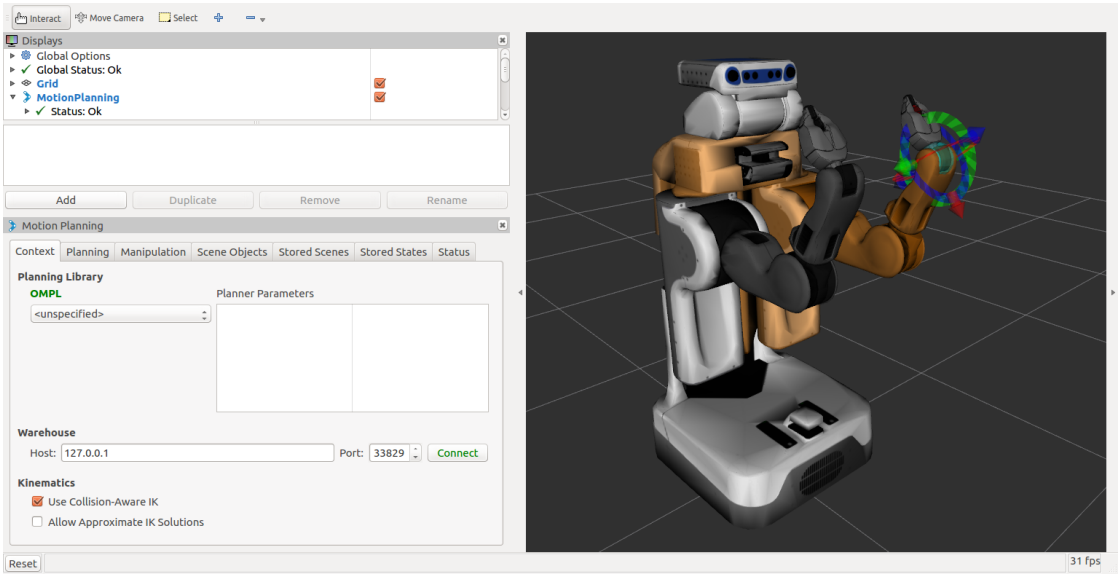

Figure 2.1: PR2 robot in RViz-based MoveIt demo mode

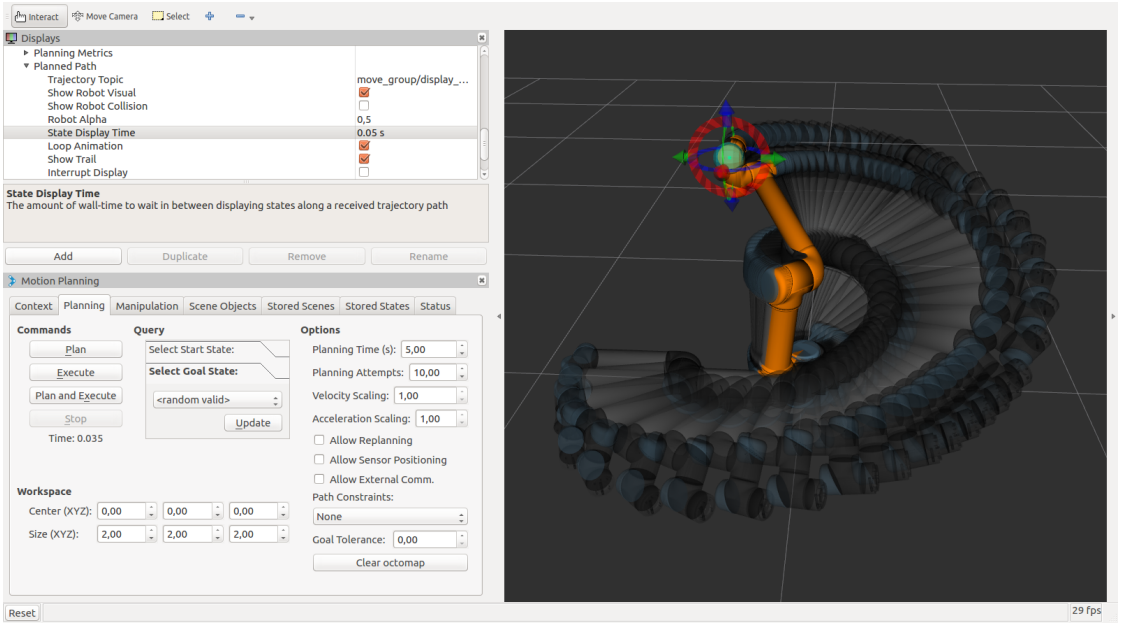

Figure 2.2: Trajectory planned for the UR5 robot using MoveIt - RViz-based MoveIt demo mode

2.1.8 Gazebo

Gazebo [6] [7] is a robot simulator which can be used in combination with ROS. Gazebo provides physics simulation and visualization.

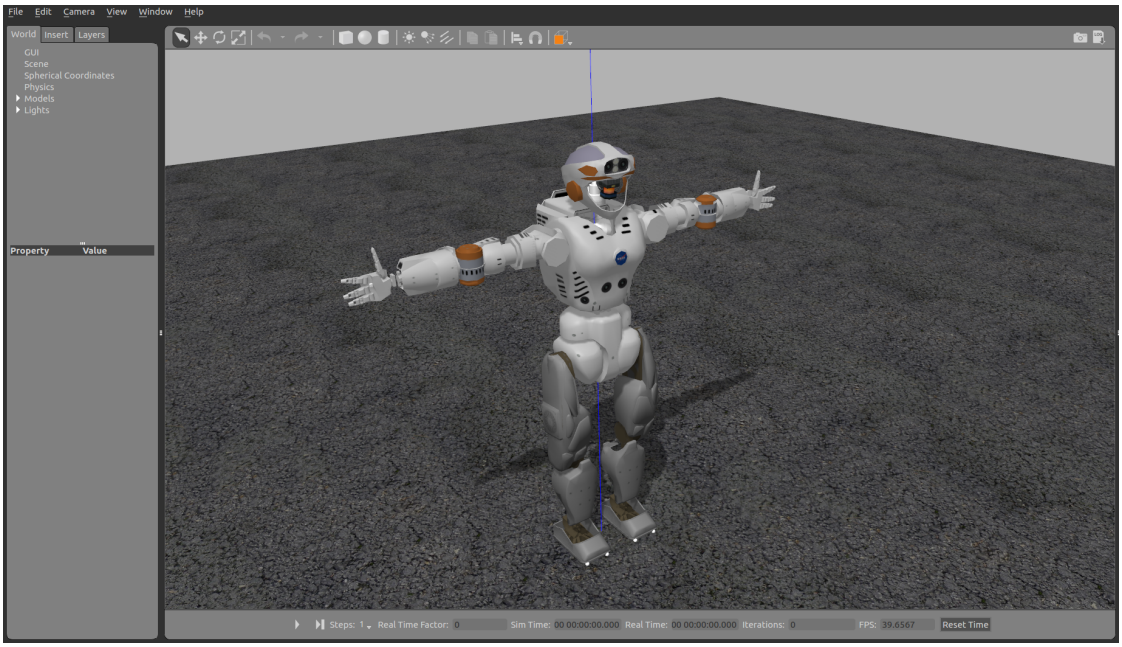

Figure 2.3: Gazebo robot simulation, with NASA Valkyrie robot

2.1.9 RViz

RViz can be used in ROS for visualization. Different components are provided for different visualization tasks. A RobotModel component can be used to render URDF models and to visualize robot poses. PointCloud receives point clouds through ROS messages and visualizes them. An InteractiveMarkers component is provided for user interaction. RViz is frequently used in combination with MoveIt. MoveIt also provides it's own MotionPlanning RViz component, which can be used to control some of the MoveIt functionality interactively.

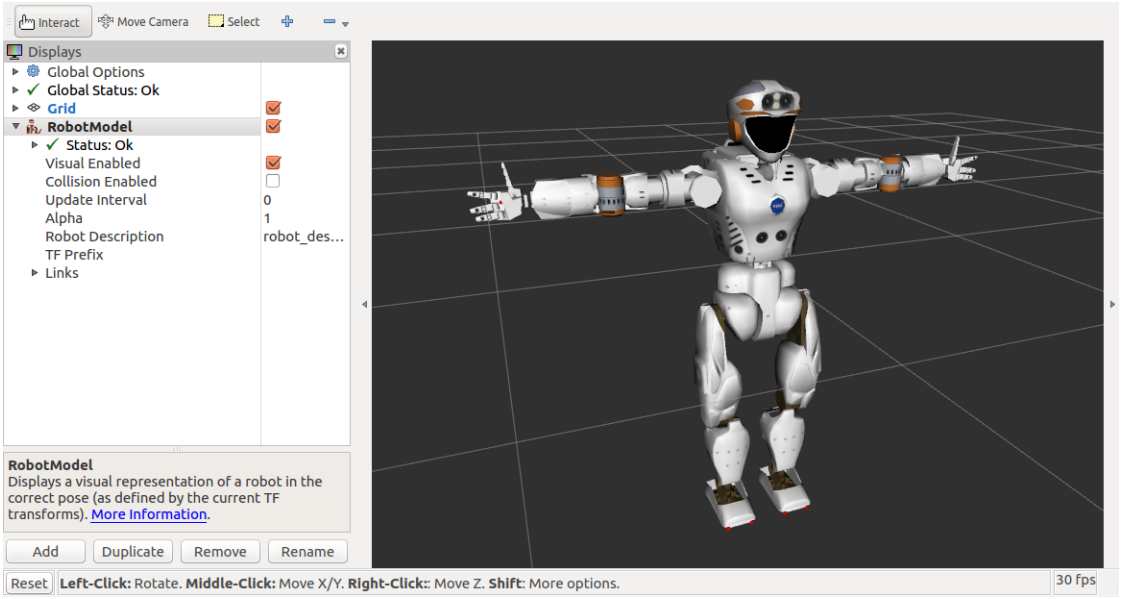

Figure 2.4: RViz, with NASA Valkyrie robot

2.1.10 URDF

In ROS, robot models are typically described via URDF (Unified Robot Description Format) files. URDF is an XML based file format, which—among other things—encodes the kinematic structure of a robot. ROS provides packages for working with URDF files. MoveIt expects robots to be defined in URDF. RViz can also visualize URDF files. Gazebo supports URDF and SDF.

2.1.11 SRDF

MoveIt uses SRDF files (Semantic Robot Description Format) to store additional information about a robot, including which parts of the robot should be used for motion planning and inverse kinematics.

2.1.12 SDF

SDF (Simulation Description Format) is used to describe Gazebo simulations. To simulate a robot in Gazebo, the robot could either be described in URDF, or in SDF. Often, the robot itself is described in URDF, and only simulation specific information is described in SDF. SDF tags can also be embedded in URDF files.

2.2 Kinematic Chains and Kinematic Trees

Virtual robot models are commonly represented as kinematic trees or kinematic chains, consisting of links and joints. Each link represents a part of the robot as a rigid body. A joint describes a connection between two links. Links can be moved in relation to each other via the connecting joints.

A link in a kinematic tree can have a single parent link and one or more child links. The kinematic tree starts at a root link and ends in one or more tip links. In a kinematic chain, each link can only have a single child link.

Joints can be of different joint types. A fixed joint describes a static connection between two links. A revolute joint can rotate one link in relation to another link around a given joint axis. A prismatic joint moves a link in relation to another link along it's joint axis. The current state of a joint is encoded in it's joint variables. The movement of a joint can be restricted by joint limits. Joint limits are usually specified as upper and lower bounds on the joint variables.

A robot state can be described in joint space by assigning values to the joint variables. A robot state can also be described in Cartesian space by specifying the positions and orientations of the links.

2.3 Forward Kinematics

Forward kinematics maps a joint-space representation of a robot state to the Cartesian-space position and orientation of an end effector link. Forward kinematics can be computed by starting at the root link, computing the relative joint transforms of the connecting joints, and concatenating the relative joint and link transforms.

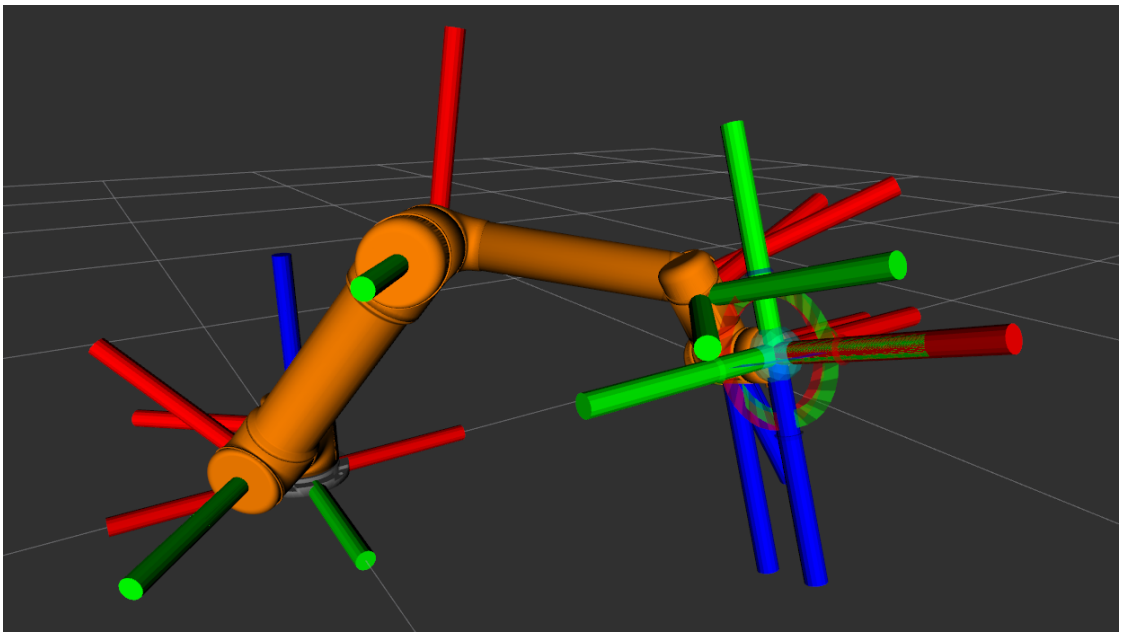

Figure 2.5: Link frames of the Universal Robots UR5 robot arm. Each frame is specified by a local coordinate system (x red, y green, z blue). Rotation axes are aligned to the y axes by convention

2.4 Inverse Kinematics

Inverse kinematics maps end effector poses to joint values. Single goal IK operates on a kinematic chain and finds joint values for a single end effector link to match a single goal pose. Multi goal IK operates on a kinematic tree and finds joint values for multiple end effector links to match multiple goal poses. Inverse kinematics can be generalized to finding joint values for which the corresponding Cartesian-space robot pose matches a set of arbitrary constraints.

In some cases, the forward kinematics equations can be reversed analytically, allowing inverse kinematics solutions to be computed directly. This approach is mainly used for single-goal inverse kinematics on simple robot structures. Robots can be specifically designed to simplify analytical inverse kinematics. For multi-goal IK and/or complex robot models, analytical solutions often become impractical and numerical approaches are preferred.

Numeric approaches iteratively improve approximate IK solutions until either a sufficiently accurate solution has been found or the search aborts.

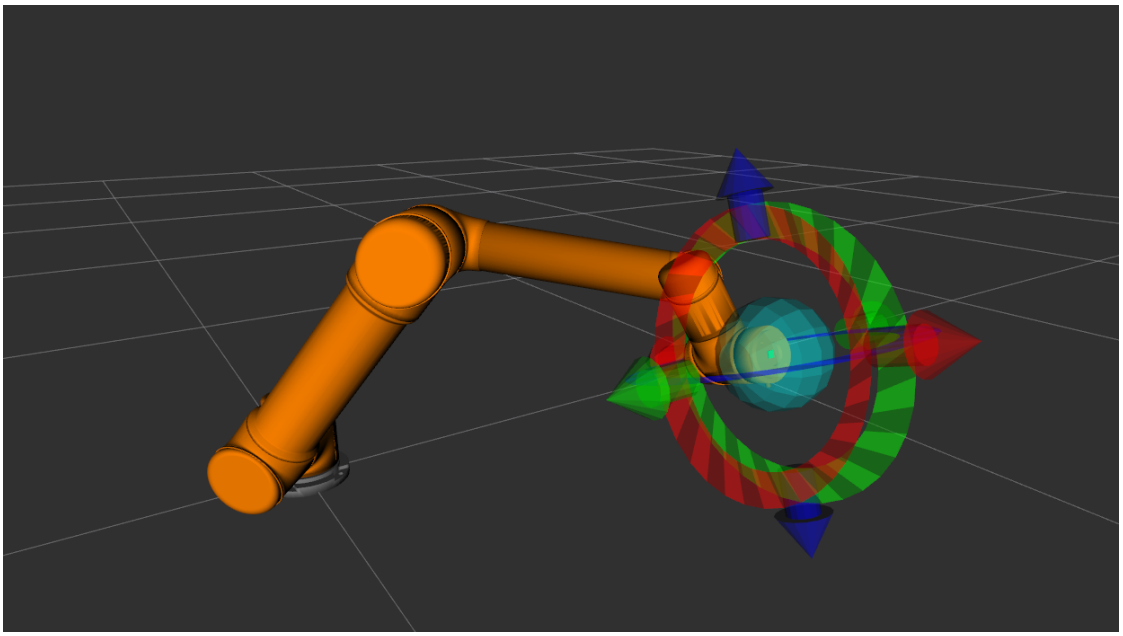

Figure 2.6: An inverse kinematics solution has been found for the UR5 robot arm. The solution is visualized in RViz. The goal pose can be controlled by the user through an interactive marker.

2.4.1 Pseudo-Inverse Jacobian Method

A popular approach for numerically solving inverse kinematics is the pseudo-inverse Jacobian method. The algorithm iteratively minimizes a multi-dimensional end-effector offset (residual).

$$minimize \quad \|f(j) - g\| \tag{2.1}$$

f: forward kinematics j: joint values g: goal pose

Each iteration first computes the Jacobian matrix of the forward kinematics function at the current joint positions, differentiating the end effector pose with respect to the joint values.

$$J(j) = (\frac{\delta p_i}{\delta j_k}(j)) = \begin{bmatrix} \frac{\delta p_1}{\delta j_1}(j) & \frac{\delta p_1}{\delta j_2}(j) & \dots & \frac{\delta p_1}{\delta j_k}(j) \\ \frac{\delta p_2}{\delta j_1}(j) & \frac{\delta p_2}{\delta j_2}(j) & \dots & \frac{\delta p_2}{\delta j_k}(j) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta p_i}{\delta j_1}(j) & \frac{\delta p_i}{\delta j_2}(j) & \dots & \frac{\delta p_i}{\delta j_k}(j) \end{bmatrix} \tag{2.2}$$

i: goal pose dimension k: joint variable index

The Jacobian matrix approximates end effector movements with respect to joint movements.

$$\Delta p = J(j) \cdot \Delta j \tag{2.3}$$

The joint value offset between the current iteration and the next iteration should move the end effector position onto the goal position.

$$g - p = J(j) \cdot \Delta j \tag{2.4}$$

To find the joint offset, the equation is reversed. The inverse of the Jacobian matrix is generally not defined, since the matrix is usually not square and can be singular. A pseudo-inverse is computed instead, which is also defined for non-square matrices, and if the the matrix is singular and no exact solution can be found, provides a best guess (e.g. according to least squares).

$$\Delta j = J(j)^{-1} \cdot (g - p) \tag{2.5}$$

The step size can optionally be adjusted by an additional line search.

The joint values are then moved by the computed joint offsets.

$$j_{n+1} = j_n + J(j_n)^{-1} \cdot (g - f(j_n)) \tag{2.6}$$

In case of joint limits, joint movements can be clipped at the joint limits.

Iterations are repeated until a stop criterion is met (e.g. until the distance between end effector pose and goal pose is below a certain threshold, or until a timeout occurs).

2.4.2 Gradient Descent

A simpler optimization method, which does not require inverting the Jacobian matrix, attempts to descend along the gradient directly. Gradient descent is a popular optimization method due to it's simplicity and low computational cost per iteration. However, depending on the optimization problem, the gradient can significantly deviate from the direction towards the optimum, in which case gradient descent performs poorly. For inverse kinematics, the pseudo-inverse Jacobian method is usually preferred.

2.4.3 Issues

Inverse kinematics problems frequently exhibit local optima, which can cause the search to converge towards incorrect solutions.

- Rotational joints can cause non-linear end effector motions. The concatenation of multiple rotational transforms can lead to local optima.
- Joint limits can cause local optima by blocking paths to the global optimum.
- In the case of multi-goal IK, the problem of reaching all goals simultaneously can lead to additional local optima.

2.4.4 Inverse Kinematics Solvers for MoveIt

MoveIt provides a plug-in interface for inverse kinematics solvers. Multiple implementations are already available. The inverse kinematics interface allows multiple IK goals for multiple end-effectors to be passed to IK solvers. However, current IK implementations are limited to single-goal IK. Currently, the interface does not support specifying goal types. All goals are passed to the IK solvers as 6 DOF poses (position and orientation). Some IK solvers support a "position_only_ik" configuration parameter (fetched during initialization from the ROS parameter server), which, if enabled, causes the solver to ignore goal orientations.

KDLKinematicsPlugin

The default inverse kinematics solver for MoveIt uses the pseudo-inverse Jacobian method. The MoveIt plugin acts as a wrapper around an inverse kinematics solver from the Orocos Kinematics and Dynamics Library [8].

LMAKinematicsPlugin

A variant of the default IK solver (KDLKinematicsPlugin) exists, which uses the Levenberg-Marquardt algorithm, interpolating between pseudo-inverse Jacobian and gradient descent using a damping factor. LMAKinematicsPlugin acts as a wrapper around an LMA based IK solver in the Orocos Kinematics and Dynamics Library [8].

TRAC_IKKinematicsPlugin

TRAC-IK [9] [10] runs a modified version of the KDL pseudo-inverse Jacobian solver and a sequential quadratic programming method in parallel. Random jumps have been added to the pseudo-inverse Jacobian solver to mitigate local minima at joint limits. Sequential quadratic programming (SQP) uses a quadratic approximation at each iteration.

2.5 Motion Planning in MoveIt

One of the core functionalities of MoveIt is motion planning. The result of a motion planning operations is a joint-space trajectory, consisting of a sequence of timed joint-space poses. When executing the trajectory, the robot successively assumes the planned intermediate poses.

Motions can either be planned from a joint-space start pose to a joint-space target pose, or by tracking the path a Cartesian-space end effector using inverse kinematics. Movements can be further restricted by user-defined constraints.

Motion planners in MoveIt implement a common plug-in interface. Different implementations are available MoveIt provides a plug-in interface for motion planners. Different implementations are available, some of which integrate external libraries, e.g. OMPL (Open Motion Planning Library) [11].

2.6 Parallelism

Modern x86 based architectures implement different types of parallelism, allowing multiple calculations to be done simultaneously.

2.6.1 Multiprocessing

Multiple parallel threads of execution can be run on multiple processing units (CPUs / CPU cores), each having it's own control unit, executing an independent instruction stream. Operating systems, program libraries and special processor instructions provide mechanisms to control thread execution and information exchange. Throughput can be increased by distributing computations across multiple CPUs / CPU cores.

Locks / Mutual Exclusion

Mutual exclusion can force a specific part of a program to be executed by only one thread at a time (to prevent race conditions, e.g. data corruption due to multiple threads modifying the same data structure simultaneously). For performance-critical applications, locks can cause significant overhead. Performance optimization in a multiprocessing context often attempts to limit explicit synchronization.

Barriers

Threads can by explicitly synchronized using barriers. If a thread reaches a barrier, which has not yet been reached by all other threads, the barrier causes the thread to wait until all other threads have reached the barrier.

2.6.2 Vectorization

SIMD parallelism (Single Instruction, Multiple Data) lets a single instruction perform operations on multiple values in parallel. Multiple components of a vector can be processed by a single vector instruction.

Instruction Sets

SSE (Streaming SIMD Extensions) features 8 128bit vector registers. SSE1 defines 32bit (single precision) floating point operations on scalars and four component vectors. SSE2 defines 64bit (double precision) floating point operations on scalars and two component vectors.

AVX (Advanced Vector Extensions) features 16 256bit vector registers and vector operations on 8D single precision or 4D double precision vectors [12].

FMA (Fused Multiply-Add) provides multiply-accumulate instructions, which allow two vectors to be multiplied and added to a third vector by a single instruction. FMA instructions can work on SSE or AVX registers [12].

Alignment

For many processor architectures, loading data into registers and storing data in memory is often faster, if the address is a multiple of the word size. While alignment has become less important on modern x86 based CPU architectures for scalar instructions, it usually is still significant when working with vector instructions. Vector instruction sets often provide different instructions for aligned and unaligned memory access. Automatic alignment ensured by the compiler is not always sufficient for vector instructions. Vector instructions often require programmers to ensure alignment manually, to be able to use fast aligned read and write vector instructions.

Processor Intrinsics

C++ compilers commonly expose CPU architecture specific instructions (such as SIMD instructions) as intrinsic functions. SIMD instructions can also be called via assembler. Compiler intrinsics allow all code to be written in C++, typically increase readability, and can allow the compiler to automatically re-order instructions for optimal pipelining.

Auto Vectorization

Loops can in some cases be vectorized be the compiler automatically. The loop is first unrolled according to vector size. Multiple iterations are executed simultaneously and each instruction processes values from multiple different iterations at the same time. Auto-vectorization requires loop iterations to be independent.

Popular compilers (e.g. GCC) provide different ways to request auto-vectorization. Auto-vectorization can be enabled globally via compiler flags. In

this case however, the compiler has to ensure correctness without being able to make additional assumptions about iteration independence or memory alignment. This often prevents efficient auto-vectorization. Some compilers also provide ways to explicitly request vectorization of a specific loop and to specify additional loop invariants. The OpenMP standard defines a "#pragma omp simd" compiler directive, which can be placed in front of a loop to requests auto vectorization, and also allows the programmer to specify explicit memory alignment hints.

Function Multiversioning

For optimal performance across different platforms and different vector instructions set, it may be desirable to provide different implementations of the same function. Providing different platform-dependent implementations of the some function can be done by manually by writing different versions of the same function and selecting the correct one at runtime. Some compilers (e.g. gcc) support built-in function multiversioning [13], allowing a function to be overloaded by processor instruction set.

2.6.3 Pipelining

Processor instructions can take more than one clock cycle to complete (instruction latency). Some of these instructions may be implemented as multiple independent steps, so that additional instructions can be issued before the first one completes. For optimal throughput, the instructions should be pipelined according to instruction latencies. This can usually be done by the compiler. How well a code section can be pipelined can vary widely, depending on interdependencies between instructions (when the result of an instruction is needed by another instruction), and depending on control flow.

2.7 Biologically Inspired Optimization Methods

2.7.1 Evolutionary Algorithms

Evolutionary algorithms are optimization methods that are inspired by biological evolution. The optimization problem is defined as a fitness function, according to which a population is evolved towards higher fitness. Each individual in the population represents a candidate solution. Over multiple generations, the individuals are modified through biologically inspired operators such as mutation and crossover, and then selected for optimal fitness.

Genetic Algorithms

In a genetic algorithm, each individual stores a candidate solution in it's genome, typically encoded as a bit string. The genome could for example consist of multiple binary integers. Each generation involves crossover, mutation and selection.

Crossover combines parts of the genome of one parent and parts of the genome of another parent to produce a child genome. Mutation can be implemented by randomly flipping bits in the genome. Finally, the fitness function is evaluated for each individual and individuals with high fitness are selected for the next generation.

Evolution Strategies

Evolution strategies store candidate solutions as floating point numbers. Mutation is performed by adding normally distributed random numbers. In some variants, each child has two or more parents, and the parent genes are randomly mixed. In other variants, each child only has a single parent. Selection is based on fitness ranking. Individuals are sorted by their fitness and the individuals with the highest fitness are selected.

2.7.2 Memetic Algorithms

Memetic algorithms combine evolution with other optimization methods. To efficiently solve a real-valued optimization problem, evolution can be used as a robust way to overcome local optima and to find a rough guess close to the global optimum, and then a gradient based optimization method can be used for fast local search and to quickly converge to an accurate solution. This approach combines the advantages of evolutionary algorithms with those of gradient based optimization methods.

2.7.3 Particle Swarm Optimization

Another population based optimization method is particle swarm optimization. The position of a particle represents a candidate solution. At each iteration, random movements are tried, and if the fitness improves, are applied to the particle. In addition to the position, each particle also has a momentum. If a random movement improves a particle's fitness, it is also added to it's momentum. Conversely, the momentum is added to the particle's position during each iteration. This allows the particle to accumulate momentum and accelerate towards the optimum and thus reach the optimum faster.

2.8 Rotation Formalisms

Rotations in three-dimensional space can be represented in different ways. Choosing a suitable rotation formalism depends on the application.

2.8.1 Euler Angles

An object can be brought into an arbitrary orientation by successively applying three axis-aligned rotations. For example, the object could first be turned left or right, then up or down, and finally around it's forward axis (yaw/pitch/roll). Euler

angles describe rotations in a way that can be interpreted by humans relatively easily. However, this representation is for most applications computationally inefficient and mathematically inconvenient.

- Correctly interpreting Euler angles depends on convention. The result depends on the order in which the three rotations are applied.
- Singularities exist, in which two of the three rotation axes align, and no parameter is available for rotating orthogonally to the first and second axis. This can lead to problems with incremental rotations and when working with derivatives.
- Geometrically correct concatenation of Euler angle rotations is not trivial and has to account for possible direction changes of the latter two rotation axes. Special care has to be taking to correctly handle singularities. Geometrically correct concatenation of Euler angle rotations is often implemented by first converting Euler angles to another representation, then performing the operation, and finally converting the result back to Euler angles.
- Rotating a vector using Euler angles requires evaluating computationally expensive trigonometric functions.

2.8.2 Rotation Matrices

The orientation of an object can be represented by three orthogonal basis vectors. These basis vectors can be composed into a 3x3 rotation matrix.

Rotating a point or vector by a rotation matrix can be performed by multiplying the rotation matrix with the vector. This makes rotation matrices the preferred rotation formalism when large numbers of points or vectors need to be transformed in a computationally efficient way, e.g. for computer graphics or collision checking.

Consecutive rotations by rotation matrices can be combined through matrix multiplication.

Rotation matrices need 9 parameters for representing three-dimensional rotations, requiring more storage space and in some cases more data transfer than other formalisms.

The basis vectors that make up a rotation matrix are unit vectors of length 1 and orthogonal. For some applications, special care has to be taken to always preserve these invariants. If for example a rotation matrix is incrementally modified over a large number of computational steps by multiplying it with other rotation matrices, numeric errors could in the worst case accumulate exponentially, until the basis vectors are not of the correct length and/or not orthogonal enough anymore. A way to prevent this is to regularly check the invariants, and if violated, renormalize and re-orthogonalize the basis vectors. Some other rotation formalisms also involve invariants, but those are usually less complex.

Concatenating rotation matrices and transforming points with rotation matrices does not involve computationally expensive trigonometric operations. However,

the matrix multiplications needed for concatenating rotation matrices are usually not as efficient as concatenating e.g. rotation quaternions.

MoveIt represents most three-dimensional orientations as rotation matrices.

2.8.3 Axis-Angle

Any orientation in three-dimensional space can be represented as a single rotation around a specific axis (Euler theorem). The axis can be encoded as a unit vector and the angle as an additional scalar, leading to a four-parameter axis-angle representation.

$$(\begin{pmatrix} x \\ y \\ z \end{pmatrix}, \phi) \tag{2.7}$$

Simple component-wise addition of two axis-angle representations does generally not lead to a geometrically correct concatenation. Rotating a point or vector by an angle around an axis requires evaluating computationally expensive trigonometric functions. While by itself computationally not very convenient, the axis-angle representation serves as a conceptual basis for other rotation formalisms.

2.8.4 Rotation Quaternions

The axis-angle representation can be brought into a mathematically and computationally more convenient form by replacing the angle with sines and cosines. Euler parameters (not to be confused with Euler angles) multiply the vector with the sine of half the angle and replace the angle with the cosine of half the angle.

$$\begin{aligned} q_w &= cos(\frac{\phi}{2}) \\ q_x &= x \cdot \sin(\frac{\phi}{2}) \\ q_y &= y \cdot \sin(\frac{\phi}{2}) \\ q_z &= z \cdot \sin(\frac{\phi}{2}) \end{aligned} \tag{2.8}$$

The four Euler parameters can be encoded in a mathematical quaternion.

$$Q = cos(\frac{\phi}{2}) + i \cdot x \cdot \sin(\frac{\phi}{2}) + j \cdot y \cdot \sin(\frac{\phi}{2}) + k \cdot z \cdot \sin(\frac{\phi}{2}) \tag{2.9}$$

Quaternion multiplication of two rotation quaternions leads to a geometrically correct concatenation of the two rotations.

Multiplying two quaternions requires 28 scalar operations, which is less than the 45 scalar operations required for multiplying two 3x3 rotation matrices. Rotating a vector by a rotation quaternion requires 30 scalar operations, rotating a vector by a rotation matrix requires only 15 scalar operations. When concatenating two transformations, each consisting of a translation and a rotation, rotation quaternions and rotation matrices require similar operation counts. Rotation matrices require 60 operations (45 + 15), while rotation quaternions require 58 operations (28 + 30).

A valid rotation quaternion is a unit quaternion. Some applications require this invariant to be maintained explicitly. One possible solution is to periodically compute and check the length (or square length) of the quaternion, and if it deviates too far from 1, re-normalizing the quaternion by scaling it by the inverse of it's length. If the error is known to be small, the inverse square root required for computing the length of the quaternion can be replaced by a linear approximation.

$$Q' = Q \cdot ((3.0 - Q \cdot Q) \cdot 0.5) \tag{2.10}$$

$Q,\ Q'$: quaternions

Rotation quaternions cover all possible rotations twice. Each rotation can be represented by two different quaternions. For an angle of $\phi = 0°$, the corresponding rotation quaternion is $0i+0j+0k+1$. For $\phi = 360°$, the quaternion is $0i+0j+0k-1$. For $\phi = 720°$, it is $0i + 0j + 0k + 1$ again.

2.9 Rotation Vectors

The axis-angle representation can be reduced to only three parameters by scaling the axis with the angle.

$$\begin{pmatrix} x \cdot \phi \\ y \cdot \phi \\ z \cdot \phi \end{pmatrix} \tag{2.11}$$

This leads to a compact, but unlike Euler angles unambiguous, representation.

Rotation vectors are commonly used to represent small relative rotations and rotational derivatives.

For infinitesimally small angles, a rotation vector is proportional to the imaginary components of a rotation quaternion.

$$\lim_{\phi \to 0} \begin{pmatrix} x \cdot sin(\frac{\phi}{2}) \\ y \cdot sin(\frac{\phi}{2}) \\ z \cdot sin(\frac{\phi}{2}) \end{pmatrix} = \begin{pmatrix} x \cdot \phi \\ y \cdot \phi \\ z \cdot \phi \end{pmatrix} \cdot \frac{1}{2} \tag{2.12}$$

2.10 BioIK

Recently, a multi-goal inverse kinematics solver has been developed at the TAMS research group, based on biologically inspired optimization methods [19] [21] [22]. The solver was implemented in C# for the Unity3D game engine.

The algorithm combines evolutionary optimization methods with particle swarm optimization. The genome of each individual encodes a joint-space robot pose as floating point numbers. Recombination randomly interpolates between parent genes. During mutation, normally distributed random numbers are added to the genes. The mutation rate (variance of the normally distributed random numbers) is adjusted according to a heuristic error function, which is proportional to the distance between the current tip frame positions and the IK goals.

Hybrid particle swarm optimization assigns a momentum to each individual. Mutations are also added to the momentums, and during reproduction, the momentums are added back to the positions that are encoded in the genomes. Updates to the momentums and position updates due to momentum are scaled randomly.

An exploitation function implements a local search method [22], which is executed for a number of high-fitness individuals (elites). The exploitation function iterates over all genes, tries a random single-gene mutation in each direction, scaled by the heuristic error function, and if the fitness increases, applies it to the genome.

The inverse kinematics problem is defined by a set of goals. Each goal defines a partial cost function. Different goal types have been defined. Position goals minimize the Euclidean distance between end effector position and goal position. Rotation goals minimize the angle between end effector orientation and goal orientation. Pose goals mix positional and orientational errors, weighted randomly and by a size-dependent angular scale heuristic. The cost functions do not have to be differentiable. Weights can be assigned to the goals, which are used to scale the partial costs. To obtain the final fitness measure, the partial costs of all goals are added together. At each generation, individuals are selected according to fitness ranking. To evaluate Cartesian-space IK goals (position, orientation, etc.), the joint-space representations stored in the genomes have to be converted to Cartesian space through forward kinematics. Most of the computation time is spent on these forward kinematics calculations. Rotations are represented as quaternions. Local joint frames and global link frames are cached and reused if not changed.

2.11 Robot Models

A number of different already existing robot models is used in this work for experiments and for comparing IK algorithms.

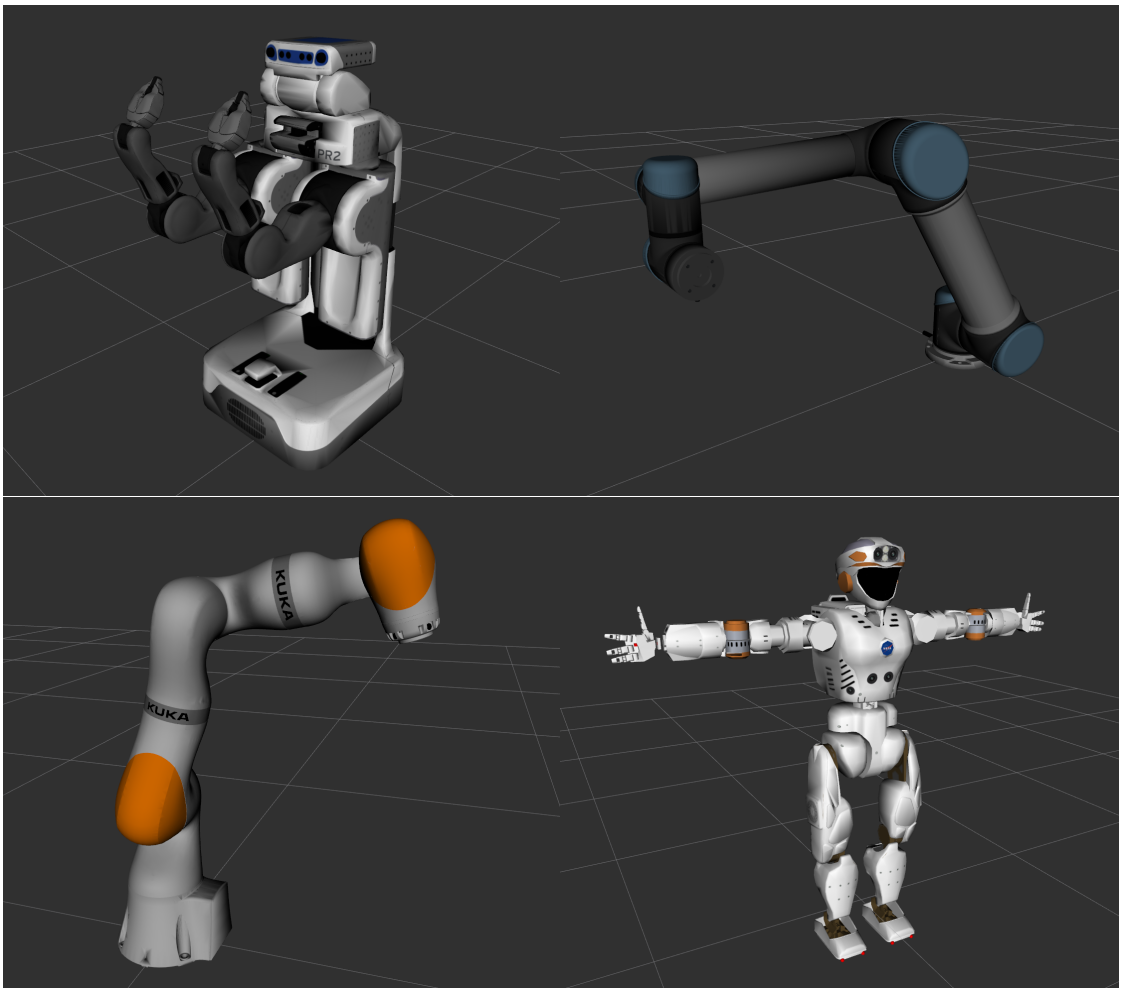

Figure 2.7: Robot models: PR2 [14] (top left), UR5 [15] (top right), KUKA iiwa [35] (bottom left), NASA Valkyrie [16] (bottom right)

Chapter 3

BioIK for ROS and MoveIt

3.1 Requirements

- Robot applications typically require high accuracy. Existing inverse kinematics solvers for MoveIt allow a default maximum error of 10^{-5}m (or 10μm) and 10^{-5}rad. For game development, accuracy is usually not as important.
- Many robotics applications require finding large numbers of inverse kinematics solutions, e.g. to accurately follow a Cartesian-space trajectory with an end effector for motion planning. Performance is therefore extremely important. In MoveIt, the runtime of an IK solver is limited by a timeout (default: 5ms).
- For robotics applications, the solver has to be able to find accurate solutions reliably. On a virtual game character, it would typically not be a problem to display sub-optimal IK solutions. On a real robot, wrong or inaccurate solutions could damage the robot or it's environment, or injure humans.

3.2 Performance Measurement

The performance of inverse kinematics solvers can be measured by success rate and average solve time. Reachable end effector poses are generated by computing forward kinematics for a random joint-space robot pose. The robot is then reset to a different random robot pose, and the inverse kinematics solver is invoked with the reachable end effector poses as inverse kinematics goals. Performance characteristics can now be measured. The test is repeated for a large number of iterations and the results are averaged. Success rate measures the percentage of IK queries that where successfully completed within a specific timeout. IK queries can also succeed before the timeout is reached. Average solve time measures the average runtime of the IK solver regardless of success or failure.

An FK-IK test framework has been implemented for MoveIt, which implements the described procedure, and is used in this work for comparing different IK solvers and implementations, and for measuring the effects of specific optimizations.

If not specified otherwise, performance measurements are taken with an inverse kinematic timeout of 5ms (MoveIt default) and a maximum error of 10^{-5}m and 10^{-5} radians.

3.3 C++ Port

The first method implemented in this work is a C++ port of the original BioIK algorithm. Only minor changes were made. Numerical precision has been increased from 32 bit single precision to 64 bit double precision. The robot model has been adapted to the MoveIt robot model, adopting MoveIt-specific features such as mimic joints.

The implementation is able to successfully solve single-goal and multi-goal inverse kinematics problems.

However, for single-goal IK, for which MoveIt already provides IK solvers, performance is below that of already available gradient-based solvers.

	Evolution	KDL	TRAC-IK
PR2	51.5%	50.6%	99.9%
UR5	24.1%	41.8%	99.7%

Table 3.1: IK success rate for the evolutionary method, KDL, and TRAC-IK, on the robots PR2 and UR5

The success rate degrades especially towards workspace boundaries.

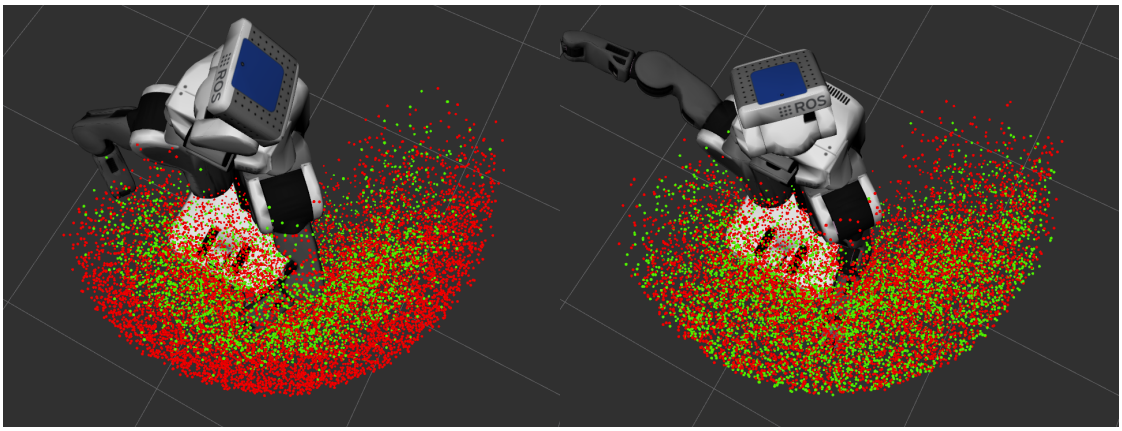

Figure 3.1: IK success/failure distribution across the workspace (PR2 robot): evolutionary method (left) and KDL (right)

3.4 Optimization

Optimizations have been made, which increase the overall performance while still maintaining the general structure of the algorithm.

3.4.1 Quadratic fitness function

The original algorithm used linear cost measures, namely Euclidean distance for positions and angles for orientations. For pose goals, positional and rotational errors were mixed randomly.

Faster convergence could be achieved by replacing linear cost measures with quadratic ones. For positions, squared vector distances are used. For orientations, the squared distance between the rotation quaternions is computed. Since rotation quaternion represent the set of all possible rotations twice, one of the quaternions is wrapped around (by negating each component) if necessary.

Usage of a quadratic fitness function already increased the performance above that of the gradient-based inverse kinematics solver provided by KDL.

	Linear Fitness	Quadratic Fitness	KDL	TRAC-IK
PR2	51.5%	57.0%	50.6%	99.9%
UR5	24.1%	43.2%	41.8%	99.7%

Table 3.2: IK success rate comparison: evolution with linear fitness, evolution with quadratic fitness, KDL, TRAC-IK, on the robots PR2 and UR5

3.4.2 Islands / Parallelization

The algorithm is parallelized by spreading the population across multiple islands and running each island independently from the other islands on a separate thread. This allows for efficient parallelization with minimal synchronization overhead, while still maintaining correctness and preventing race conditions. Once a sufficiently good solution has been found, evolution is stopped on all islands, and the best solution is returned.

	Original	Modified 1 Thread	Modified 4 Threads	KDL	TRAC-IK
PR2	51.5%	57.0%	77.4%	50.6%	99.9%
UR5	24.1%	43.2%	53.4%	41.8%	99.7%

Table 3.3: IK success rate comparison: original algorithm, modified evolution on a single island, modified evolution on 4 islands and 4 concurrent threads, KDL, TRAC-IK, on the robots PR2 and UR5

Overall performance is further increased. However, at the workspace boundary, the success rate is still relatively low.

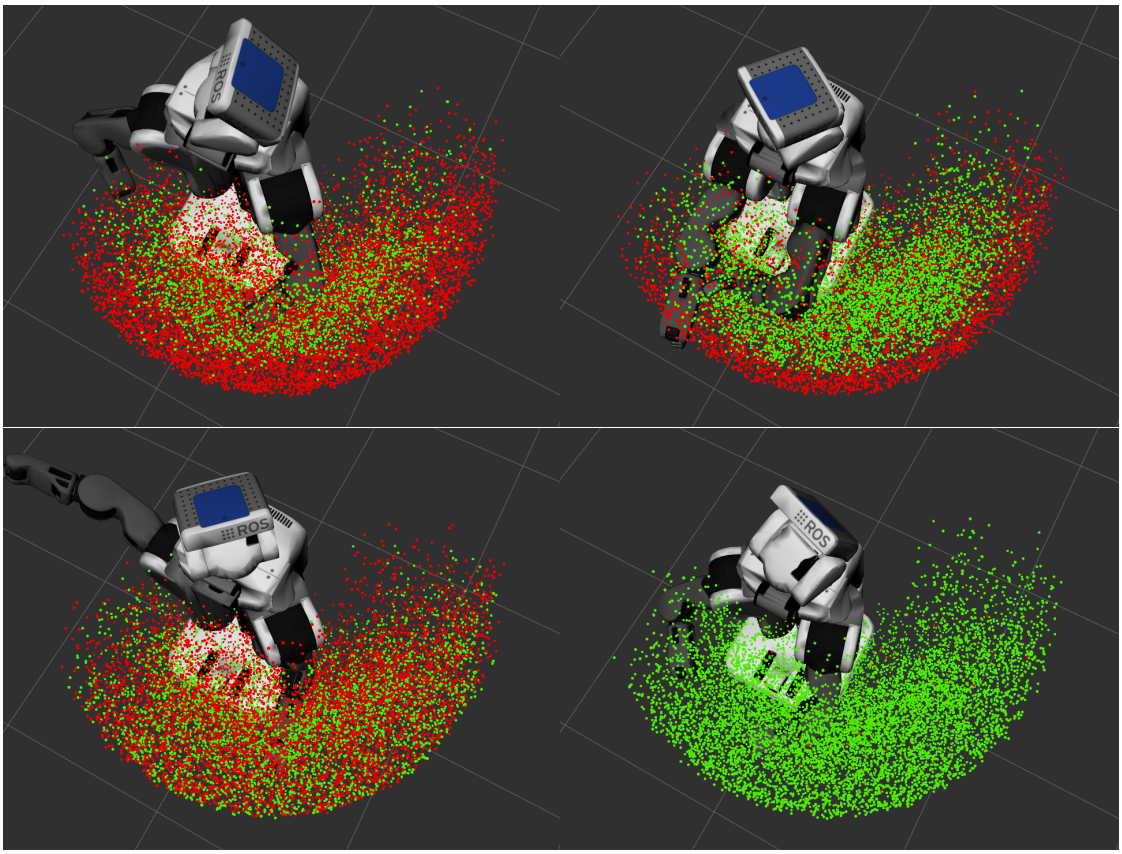

Figure 3.2: IK success/failure distribution across the workspace (PR2 robot): first version of the evolutionary algorithm (top left), modified evolution on 4 threads (top right), KDL (bottom left), TRAC-IK (bottom right)

3.5 Gradient Based Methods

Several gradient based optimization methods have been implemented for comparison and to identify candidates for memetic optimization.

Each method can optionally be run on multiple threads. Initial guesses from MoveIt are used on the first thread or if multi-threading is disabled. The other threads are initialized randomly.

3.5.1 Pseudo-Inverse Jacobian Method

A custom multi-goal version of the popular pseudo-inverse Jacobian method has been implemented for comparison. The Jacobian matrix is computed analytically. Support is limited to pose goals.

	Jac 1 T	Jac 4 T	KDL	TRAC-IK
PR2	67.3%	82.5%	50.6%	99.9%
UR5	87.9%	95.5%	41.8%	99.7%

Table 3.4: IK success rate: custom Jacobian solver on 1 thread, custom Jacobian solver on 4 threads, KDL, and TRAC-IK, on the robots PR2 and UR5

3.5.2 Gradient Descent

Gradient descent methods have been implemented by numerically differentiating the fitness function that is used by the evolutionary algorithm. The implementation supports multi-goal IK and all available goal types.

During each iteration, the fitness function is differentiated with respect to each joint variable by offsetting each joint variable by a small increment, re-evaluating the fitness function, and computing the difference quotient. The resulting vector is normalized and used as the descent direction. Next, a step size is estimated by differentiating and linearly extrapolating along the descent direction.

$s = f \div \frac{df}{dx}$, s: step size, f: fitness, $\frac{df}{dx}$: gradient

The current position is then moved by the step size along the descent direction and the new joint positions are clipped at joint limits.

It may seem reasonable to reject position updates with increasing cost. However, this could cause the optimization to be stuck at a local optimum due to deterministic descent direction and step size. Local minima can be mitigated through random resets. Alternatively, steps can be always accepted, regardless if the solution improves or not, which proved to be most effective. The custom gradient descent solver showed similar performance as the custom pseudo-inverse Jacobian solver. On the PR2, gradient descent performs slightly better, on the UR5, the pseudo-inverse Jacobian method performs slightly better.

	GD	GD R	GD C	GD C 4	Jac 1	Jac 4	KDL	TRAC-IK
PR2	7.7%	36.9%	67.8%	85.0%	67.3%	82.5%	50.6%	99.9%
UR5	1.4%	10.9%	83.8%	90.5%	87.9%	95.5%	41.8%	99.7%

Table 3.5: IK success rate (on the PR2 and UR5 robots): gradient descent, gradient descent with random resets, gradient descent with continuation, gradient descent with continuation and multi-threading on 4 threads, custom pseudo-inverse Jacobian solver on 1 thread, custom pseudo-inverse Jacobian solver on 4 threads, KDL, TRAC-IK

3.5.3 CppNumericalSolvers

In addition to the custom-implemented methods specifically optimized for solving inverse kinematics problems, several generic gradient based optimization methods from the CppNumericalSolvers library were systematically tested. The fitness function is used as the cost function. Some solvers support box constraints. These are used for specifying joint limits.

	UR5	PR2
BFGS	92.5	13.9
Conjugate GD	35.4	13.5
Gradient Descent	35.6	6.9
L-BFGS-B	0.4	1.7
L-BFGS	0	0.1
Nelder-Mead / Simplex	30.7	15.8
Newton Descent	18.1	3.1

Table 3.6: IK success rate on the PR2 and UR5 robots using several implementations from the CppNumericalSolvers library

3.6 Neural Networks

Artificial neural networks [17] were considered for local search and memetic optimization. Two neural network based approaches were implemented and tested. FANN (Fast Artificial Neural Network Library) is used for implementation.

A simple approach to neural network based inverse kinematics is to learn a mapping from end effector poses to joint angles. However, this usually leads to inaccurate solutions. Furthermore, the network fails to learn correct solutions at discontinuities (small end-effector movements can in some cases lead to large jumps in joint-space positions).

A different approach has been developed, which learns relative offsets. The network receives the current joint-space robot pose as well as the Cartesian-space differences between tip frame and effector poses as inputs. Joint-space offsets are used as outputs. The network is trained to return joint-space offsets that minimize goal offsets. During inference (to solve an inverse kinematics problem), the network is run in a feedback loop together with a forward kinematics solver, inspired by recurrent neural networks. If the goal offsets are very small, the inputs are scaled up and the outputs are scaled down again to reduce the effect of constant errors. During each iteration, the network tries to bring the end effector as close to the goal as possible. Iterations are repeated until a sufficiently good solution is found. This approach allows arbitrarily accurate inverse kinematics solutions to be found using artificial neural networks.

Compared to gradient based methods, the idea was to let the neural network learn problem-specific heuristics about non-linearities, about how to resolve redundancies, and about how to avoid joint limits.

The main drawback is that the network has to be trained for each robot setup. The performance varied between different training runs (around 20% to 40% for the PR2, a few minutes training, and standard 5ms timeout and 10^{-5} maximum error).

3.7 Modified Algorithm

The BioIK algorithm has been re-designed for high accuracy and improved performance, according to the robotics-specific requirements mentioned above.

	PR2	UR5	Valkyrie arm	Valkyrie foot	iiwa
BioIK 2	100.00%	99.93%	99.93%	100.00%	99.93%
TRAC_IK	99.91%	99.29%	99.64%	99.98%	99.88%
KDL	53.10%	41.70%	41.13%	91.68%	51.62%

Table 3.7: IK success rate: re-designed evolutionary algorithm, TRAC_IK, KDL

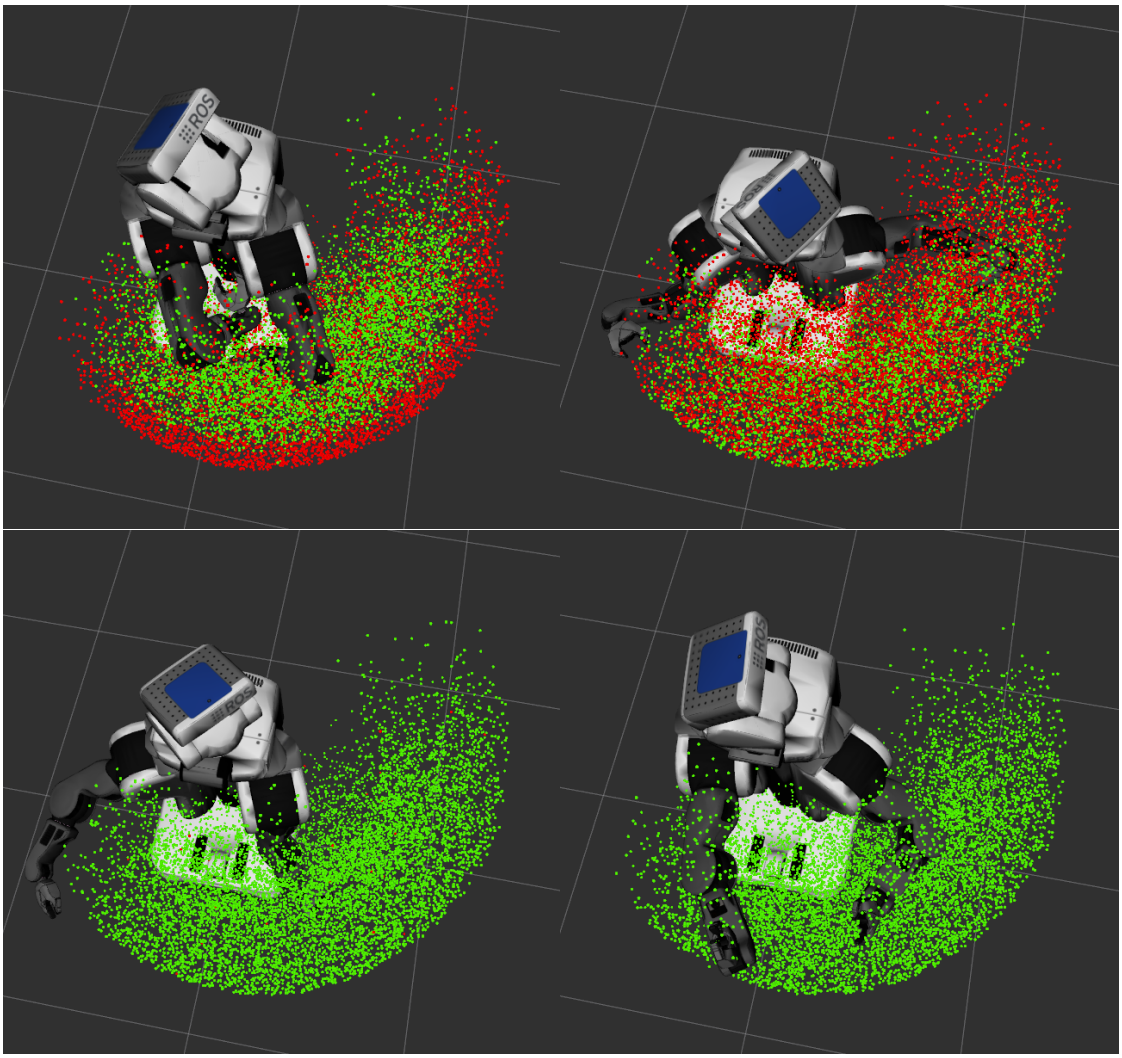

Figure 3.3: IK success/failure distribution across the workspace (PR2 robot): BioIK 1 (top left), KDL (top right), TRAC_IK (bottom left), BioIK 2 (bottom right)

3.7.1 Genome

The genome encodes joint-space robot poses as double precision floating point numbers. Only variables that are used as parameters by an IK goal and variables that influence the pose of a end effector are included (mainly to allow for optimized memory access patterns, see Implementation).

3.7.2 Mutation

Mutations are scaled by a random exponential term. The random number used for generating the exponential term is only generated once per individual offspring and stays the same for all genes of the same individual.

$$m = r \cdot 2^{-s} \tag{3.1}$$

m: mutation r: normally distributed random number
s: random integer, generated once per individual offspring

For a single gene, the random exponential term leads to a similar distribution as would be the result of randomly mutating bits of an integer. While large mutations are still possible, many small mutations are generated as well, allowing accurate results to be found efficiently.

To efficiently find accurate solutions to non-linear optimization problems, an optimization algorithm should ideally be able to converge to an optimum exponentially, cutting the error to a fraction of it's prior value during each iteration. The exponential scaling term allows this to happen by ensuring that, for reasonable mutation sizes, for each mutation a roughly half-sized mutation is possible with a similar probability.

Other possible solutions for allowing high accuracy would be monotonically decreasing scales or heuristic error functions.

After each successful mutation which lead to a fitness increase, a monotonically decreasing scale could be reduced to a fraction of it's prior value. However, this might lead to two issues. First, it would depend on a problem-specific parameter, which would have to be manually adjusted for optimal performance. Secondly, it could cause evolution to be stuck at local minima. Once the mutation scale would be decreased at a local minimum, it could become impossible to leave the local mimum again.

A heuristic error function could also be used to adjust mutation scales. However, for optimal performance, this would require a heuristic error function to be implemented for each goal type. The heuristic error function would also have to account for non-linearities. If, for example, a robot arm consisting of several revolute joints is fully extended and the IK goal is offset inwards, an optimal mutation would have to be much larger than if the goal would be offset sidewards, so an optimal heuristic error function for position goals would be direction dependent. The

heuristic error function of the first BioIK algorithm was not direction dependent, causing slow convergence at workspace boundaries.

In many cases, two or more genes have to mutate simultaneously to increase fitness. For example, reducing the distance between an end effector position and a goal position for a fully or almost fully extended revolute joint robot arm can require simultaneous reciprocal mutations to multiple joints. Only mutating one of the joints would increase the distance, e.g. moving too much sidewards, even if the inwards or outwards directed part of the movement would be correct. If the random mutation scale would be generated per-gene (or if integer genes would be mutated by flipping random bits), most mutations would move a single joint by a significantly larger amount than the other joints, making similarly scaled simultaneous mutations to multiple genes very unlikely. The problem could be solved using a large population size. A first mutation would then be allowed to temporarily decrease the fitness, until a second mutation increases the fitness again. However, this would increase the computational cost substantially. Instead, the issue is solved by generating random mutation scales only once for each individual offspring and keeping the mutation scale the same for all genes of the same individual.

	PR2	UR5
BioIK 2, with memetics, per-individual mutation size	100.00%	99.93%
BioIK 2, no memetics, per-individual mutation size	100.00%	99.86%
BioIK 2, no memetics, per-gene mutation size	76.38%	35.85%
TRAC_IK	99.91%	99.29%
KDL	53.10%	41.70%

Table 3.8: IK success rate comparison for per-gene and per-individual random mutation sizes

3.7.3 Selection

At each generation, the best individuals are selected according to the fitness function.

The algorithm can distinguish between primary and secondary objectives. If goals have been specified as secondary objectives, individuals are first pre-selected according to the secondary objectives, and then the final selection is made according to the primary objectives. For a fixed number of survivers after pre-selection, the optimization could be stuck at points where primary and secondary objectives cancel each other out, if the number of survivers would be too small. If the number of survivers would be too large, the secondary objectives would be ignored. So for a fixed number of survivers, the number of survivers would have to be configured manually. Therefore, a random number of survivers is used for pre-selection.

3.7.4 Islands

The population is spread across multiple islands. Evolution on each island is run independently from the other islands on a separate thread. When a solution is found, evolution is stopped on all islands and the best result is selected.

	PR2	UR5
BioIK 2, 4 islands (4 threads)	100.00%	99.93%
BioIK 2, 2 islands (2 threads)	100.00%	99.81%
BioIK 2, 1 island (1 thread)	99.99%	99.34%
TRAC_IK (2 threads)	99.91%	99.29%
KDL (1 thread)	53.10%	41.70%

Table 3.9: IK success rate comparison for different numbers of islands and threads

3.7.5 Species and Wipeouts

Evolution could be temporarily stuck at a local optimum. If the fitness of the local optimum is only slightly worse than that of the global optimum, and/or if the local optimum is significantly larger than the global optimum, it can be relatively unlikely for a random mutation to move an individual from the local optimum close enough to the global minimum for the fitness to improve and for the search to continue towards the hard-to-find global optimum. Wipeouts are used to reliably move out of local optima.

Each island is home to two competing species. Individuals are only allowed to mate within their own species. Among the two species that are living on each island, only the less fit species can be wiped out, measured by the fitness of the fittest individual within each species. A wipeout re-initializes all individuals within a species from a randomly generated genome. A wipeout is triggered if a species fails to improve for several generations, or with low probability at random. Evolving two competing species on each island and only wiping out the less fit species prevents good solutions from being wipeout out accidentally.

	PR2	UR5
BioIK 2, with wipeouts	100.00%	99.93%
BioIK 2, no wipeouts	95.39%	96.42%
TRAC_IK	99.91%	99.29%
KDL	53.10%	41.70%

Table 3.10: IK success rate comparison: wipeouts

3.7.6 Initialization

In MoveIt, initial guesses are provided for inverse kinematics queries. If multiple solutions are possible, applications generally expect a solution that is close to the initial guess to be returned. To allow for fast convergence towards a solution that is close to the initial guess, the genes of all individuals are initialized with the joint values that were provided as the initial guess. If a bad initial guess is given, wipeouts are eventually triggered, which randomly re-initialize the population.

Performance can be further increased by only initializing the population on the first island with the initial guess and initializing the populations on the other islands randomly. However, this can cause possible solutions close to the initial guess to be missed. Therefore, all islands are initialized with the initial guess, despite a small performance decrease.

	PR2	UR5
BioIK 2, initial guess	100.00%	99.93%
BioIK 2, random initialization	100.00%	99.97%
TRAC_IK	99.91%	99.29%
KDL	53.10%	41.70%

Table 3.11: IK success rate comparison: initial guess vs. random initialization

3.7.7 Termination

The algorithm terminates if a sufficiently good solution is found, or if a timeout occurs. Whether a solution is good enough could be determined by a fitness threshold. For compatibility, and to allow performance to be accurately compared with other solvers, special acceptance criteria are implemented for pose goals, position goals, and orientation goals. For other goal types, the individual cost of each goal is compared to the square of the maximum allowed error (assuming most cost functions to be roughly quadratic).

3.7.8 Particle Swarm Optimization

Hybrid particle swarm optimization is also used in the modified algorithm.

In addition to the genome, each individual also has a momentum. The momentum has the same dimensionality as the genome.

In the first version of the BioIK algorithm, two real-valued random scalings were applied to each component of each individual's momentum at each generation.

$$\begin{aligned} g' &= g + m \cdot r_1 \\ m' &= m \cdot r_2 \end{aligned} \tag{3.2}$$

g: genome vector m: momentum vector r_1, r_2: random vectors

Scaling each component differently can cause the momentum to lose direction. This is avoided by choosing a random scale only once per individual. If a good direction has been found, optimization can efficiently continue into the same direction, until a good position is found along the direction of the momentum, causing the error to become mostly orthogonal to the momentum, and the momentum can be discarded and re-initialized. Also, the direction of the momentum can be refined over multiple iterations more efficiently. Per-gene randomization could prevent mutation randomness from being smoothed out enough.

Optimal momentum depends on position. Some position updates can invalidate the momentum. For example, after a large random mutation, the local fitness landscape and the relative directions towards the optima could be completely changed. Also, if an individual has accumulated momentum in the rough direction towards an optimum and has reached a good position along the direction of the momentum, the error would now be mostly orthogonal to the momentum. In these cases, it could be most effective to completely remove all momentum. However, random real-valued scales would never be exactly zero, and if the scales would be different for each dimension, it would be very unlikely for all scales to be small enough simultaneously.

Instead, a small integer is used for scaling, which is generated only once per individual. For each individual offspring, an integer from 0 to 2 is selected and before the mutation is applied, the momentum is scaled by the random integer. The momentum is thus either kept unchanged, doubled, or removed.

	PR2	UR5
BioIK 2, with particle swarm optimization	100.00%	99.93%
BioIK 2, no particle swarm optimization	99.41%	91.24%
TRAC_IK	99.91%	99.29%
KDL	53.10%	41.70%

Table 3.12: IK success rate comparison: particle swarm optimization

3.7.9 Memetics

Gradient based optimization is added for fast local search. After running evolution for a number of generations, a gradient based method is run on the best individual of each species. Local search stops either if it fails to further improve the solution, or if a fixed maximum number of iterations is reached. Afterwards, evolution continues. Gradients are computed by numerically differentiating the fitness function with respect to joint values. Different gradient based optimization methods have been tested.

	PR2	UR5	Vk arm	Vk foot	iiwa
BioIK 2, quadratic memetics	100.00%	99.93%	99.93%	100.00%	99.93%
BioIK 2, linear memetics	99.99%	99.90%	99.94%	99.99%	99.91%
BioIK 2, LBFGS-B memetics	99.99%	99.90%	99.91%	100.00%	99.92%
BioIK 2, no memetics	99.98%	99.75%	99.50%	99.99%	99.71%
TRAC_IK	99.91%	99.29%	99.64%	99.98%	99.88%
KDL	53.10%	41.70%	41.13%	91.68%	51.62%

Table 3.13: IK success rate: no memetics, linear memetics, quadratic memetics

The linear method is based on the custom gradient descent method described above.

Quadratic memetics uses a variant of the custom gradient descent method with quadratic step size estimation. The fitness landscape is approximated along the gradient by a parabola via numeric differentiation and the distance to the extremum is calculated.

$$\begin{aligned} v_1 &= f_2 - f_1 \\ v_2 &= f_3 - f_2 \\ v &= (v_1 + v_2)/2 \\ a &= v_2 - v_1 \\ s &= v/a \end{aligned} \tag{3.3}$$

f_1, f_2, f_3: three fitness values, sampled at three close points along the gradient
s: step size scale (as a multiple of the step size used for sampling f_1, f_2 and f_3)

LBFGS-B based memetics uses an LBFGS implementation from the CppNumericalSolvers library.

The final version of the algorithm uses the custom quadratic gradient descent method.

3.7.10 Extrapolated Forward Kinematics

Most of the computation time is spent on calculating forward kinematics. First, for each joint that has been moved, a local joint transform has to be computed. For some joint types (e.g. revolute joints), this involves computationally expensive trigonometric operations. Then, local joint transforms and local link transforms have to be concatenated, not just requiring computationally simple component-wise vector operations, but also computationally more heavy matrix or quaternion multiplications. To save computation time, forward kinematics is extrapolated for most mutations.

First, an exact forward kinematics solution is calculated for the best individual of each species. Next, positions and orientations are differentiated with respect to joint variables at the joint positions encoded in the best individual's genes. Evolution and memetrics are then run for a few generations. Forward kinematics is not re-computed for each mutation. Instead, the forward kinematics problem is extrapolated using the gradients and the last full forward kinematics computation. After running evolution and memetics for a few generations using extrapolation, another exact forward kinematics solution is computed and the gradients are re-initialized.

The extrapolator uses the Jacobian matrix of the forward kinematics problem, consisting of first order derivatives of the end effector positions and orientations with respect to joint values. The Jacobian matrix is multiplied with the joint offsets and the result is added to the last known end effector poses.

3.8 Goal Types

Several inverse kinematics goal types were developed. Each goal defines a partial cost function. The fitness function is obtained through summation of the goal costs.

Position Goal

The position goal tries to match the position of an end effector with a goal position. The cost function is defined as the square distance between end effector position and goal position.

$$c = \|P_E - P_G\|^2 \tag{3.4}$$

P_E: end effector position P_G: goal position c: cost

Orientation Goal

The orientation goal tries to match the orientation of an end effector with a goal orientation. The cost function computes the square distance between two rotation quaternions. Rotation quaternions represent the set of all rotations twice. $ai + bj + ck + d$ represents the same orientation as $-ai - bj - ck - d$. Therefore, the distance is computed twice, once with the positive and once with the negative of the second rotation quaternion, and the minimum is selected.

$$c = min(\|Q_G - Q_E\|^2, \|Q_G + Q_E\|^2) \tag{3.5}$$

Q_E: end effector rotation quaternion Q_G: goal rotation quaternion c: cost

Rotational distances between rotation quaternions can also be computed via the dot product. The angle between two normalized rotation quaternions is $a = \text{acos}(Q_1 \cdot Q_2)$. A distance measure which does not require trigonometric operations can be defined as $d = 1 - (Q_1 \cdot Q_2)$. However, these methods can be less stable if the quaternions are not exactly normalized. For example, the dot product could become slightly larger than one, in which case the arcus cosine in the first formula would be undefined, and the second formula would compute a negative distance. Also, if denormalization of the quaternions is proportional to their magnitudes, the results could be inverted. For two zero-rotation quaternions, the result would be 0, but for slightly larger (proportinally denormalized) quaternions, the dot product could become larger than 1, and the result could become less than zero. This can be the case even for very small errors, since $cos'(0) = 0$ and $\lim_{\alpha \to 0} \text{acos}(\alpha)' = -\infty$. These issues are usually solved by explicitly normalizing the quaternions before computing the dot product. However, this would involve calculating computationally expensive square roots.

Also, if the dot product is used as a distance measure, the gradient becomes zero if the angle between both quaternions is 180°. If the minimum square distance

is used instead, the gradient is always large if both quaternions are pointing away from each other, ensuring fast divergence away from incorrect solutions. This would also be the case if the square angle would be used, but computing the exact angle via trigonometry would be computationally more expensive.

	PR2	UR5	Vk arm	Vk foot	iiwa
Square distance	100.00%	99.93%	99.93%	100.00%	99.93%
Norm and dot product	100.00%	99.83%	86.30%	97.21%	99.73%
Square angle	99.95%	97.08%	98.97%	100.00%	98.77%
TRAC_IK	99.91%	99.29%	99.64%	99.98%	99.88%
KDL	53.10%	41.70%	41.13%	91.68%	51.62%

Table 3.14: IK success rate comparison: rotational distance measures

Pose Goal

The pose goal tries to match both position and orientation. Position and rotation errors are computed the same way as for position and orientation goals, and are then added to obtain the combined pose goal error. The weighting of position and orientation errors can be adjusted via a rotation scale parameter.

$$c = \|P_E - P_G\|^2 + min(\|Q_G - Q_E\|^2, \|Q_G + Q_E\|^2) \cdot s_r^2 \tag{3.6}$$

P_E: end effector position P_G: goal position
Q_E: end effector rotation quaternion Q_G: goal rotation quaternion
s_r: rotation scale

Line Goal

The line goal tries to move the position of a link onto a line. The line is specified via a point p and a direction vector d.

The cost function computes the square distance between the line and the link position. A plane is constructed at the point p with the line direction d as it's normal. The link position is then projected onto the plane and the square distance between the point p and the projected link position is computed.

$$c = \|f - d \cdot (d \cdot (f - p)) - p\|^2 \tag{3.7}$$

c: cost p: point on the line d: line direction f: link position

Minimal Displacement Goal

The minimal displacement goal tries to keep each joint variable as close as possible to the last robot pose (the joint values provided as the *initial_guess* by MoveIt). The cost function is computed as a sum of squared distances. For each joint variable, the difference between it's current position and the initial guess is computed. The differences are weighted by the reciprocals of the maximum joint velocities. Joint velocities are specified in the URDF robot model and are accessed via the MoveIt RobotModel interface. The cost function then squares each scaled distance and computes the sum.

$$c = \|j - i\|^2 \tag{3.8}$$

j: current joint positions i: initial guess

Center Joints Goal

The center joints goal tries to keep all joints at the center between their joint limits. The cost function is computed as a sum of squared errors.

$$c = \|j - \frac{h + l}{2}\|^2 \tag{3.9}$$

j: current joint positions h: upper joint limits l: lower joint limits

Avoid Joint Limits Goal

The avoid joint limits goal tries to keep each joint variable within the center half of each joint's joint limits. The cost function is similar to that of the CenterJoints, but the center half of each joint's range is ignored.

$$c = \sum_{i=1}^{N}((|j_i - \frac{h_i + l_i}{2}| \cdot 2 - \frac{h_i - l_i}{2})^2) \tag{3.10}$$

j: current joint positions N: number of active joint variables

h: upper joint limits l: lower joint limits

Joint Variable Goal

The joint variable goal tries to match the value of a joint variable with a specified goal. The cost function is computed as a squared difference.

$$c = (j - g)^2 \tag{3.11}$$

j: current joint position g: goal joint position

Look At Goal

The look-at goal aligns a link axis with the direction towards a goal position. The cost function rotates the axis by the link orientation, computes the direction from the link position to the goal position, and finally calculates the square distance between the directions.

Minimum Distance Goal

The minimum distance goal tries to keep the position of a link away from the goal's position by at least a minimum distance.

Maximum Distance Goal

The maximum distance goal tries to keep the position of a link within a maximum distance to the goal's position.

Direction Goal

The direction goal tries to match a link axis with a goal direction. The axis is transform by the link's current orientation and then the square distance between the axis and the goal direction is computed.

Link Function Goal

The link function goal evaluates a user-specified cost function for the link's current position and orientation.

Joint Function Goal

The joint function goal tries to meet a user-specified joint variable constraint. The constraint is defined through a function, which receives joint variable positions and can modify them. The cost function is defined as the sum of square errors between the current joint values and the results of the constraint function.

Side Goal

The side goal tries to keep a link-space direction vector and a goal direction pointing away from each other. The cost function rotates the link-space direction by the link orientation, computes the dot product with the goal direction, cuts off negative values, and computes the square.

Cone Goal

The cone goal tries to keep a link-space direction vector with a cone specified by a goal direction and a maximum angle. The cone goal can also simultaneously match the link position with a goal position.

Touch Goal

The touch goal tries to touch the surface plane of a half space with the link's collision model without intersecting the half space. The plane is specified via a point on the plane and a normal.

Different collision detection methods are used depending on the type of the link's collision model. For geometric primitives, implementations from the Flexible Collision Library (FCL) [18] are used.

For collision meshes, different methods have been implemented and tested. Using the FCL collision model provided by MoveIt was relatively slow. FCL provides fast collision detection methods for convex shapes, but MoveIt does currently not distinguish between convex and concave shapes and always builds a relatively slow bounding volume hierarchy out of individual triangles, which should only require $O(log(n))$ complexity, but the implementation apparently comes at a relatively high cost per iteration. Manually iterating over all vertices and computing the dot product between plane normal and vertex position was for typical robot models already faster, despite theoretically higher complexity of $O(n^2)$.

To further improve the performance, a fast method was implemented, which has only $O(1)$ complexity for small mutations. The method walks on the convex hull of the collision mesh towards the minimum.

The problem must be convex, even for concave objects. The touch goal searches for a point on the object for which the dot product with the normal is minimal. If a point is part of the object, but does not lie on the convex hull, it can be represented as a linear combination of two points that lie on the convex hull. Thus, for any direction vector, the dot product with one of the two points that lie on the convex hull must be at least as small as the dot product with the point that does not lie on the convex hull. So for each possible solution that does not lie on the convex hull, another solution exists which is at least as good and does lie on the convex hull. Therefore, for this problem, it is sufficient to only search for solutions on the convex hull of the collision model.

Solutions are temporarily stored and used as the initial guess for the next query. For small mutations, the vertex with the smallest dot product with the

plane normal remains the same and the amortized complexity is $O(1)$. This method performs significantly better than the other two methods.

Balance Goal

The balance goal tries to keep the robot's center of gravity above the goal position. To compute the current center of gravity of the robot, mass and center of gravity of each link is fetched from the URDF model and a weighted average is computed of the centers of gravity weighted by mass. The cost function then projects the center of gravity and the goal position onto a plane orthogonal to a user-specified gravity vector and computes the square distance between the projected goal position and the projected center of gravity.

3.9 Implementation

3.9.1 Goal Types

The goal types are implemented as C++ classes. All goal classes are derived from a common *Goal* base class and implement an *evaluate* method and a *describe* method. Information is exchanged via a *GoalContext* object, allowing the interface to be extended at a later time without breaking the API. The *evaluate* method is called after each mutation and returns a fitness measure for the current joint values and link poses. Which joints and links a goal depends on is queried by the IK solver during initialization by calling the *describe* method.

BioIK can be easily extended by implementing new goal classes. Additional goal classes do not have to be implemented withing the BioIK package, but can be implemented within the package that calls the BioIK solver.

For goals which only depend on a single link, a *LinkGoalBase* class is provided, which implements the *describe* method. Most link goals are derived from *LinkGoalBase*. To implement a new link goal type, only the *evaluate* method has to be implemented, if the new goal type is derived from *LinkGoalBase*.

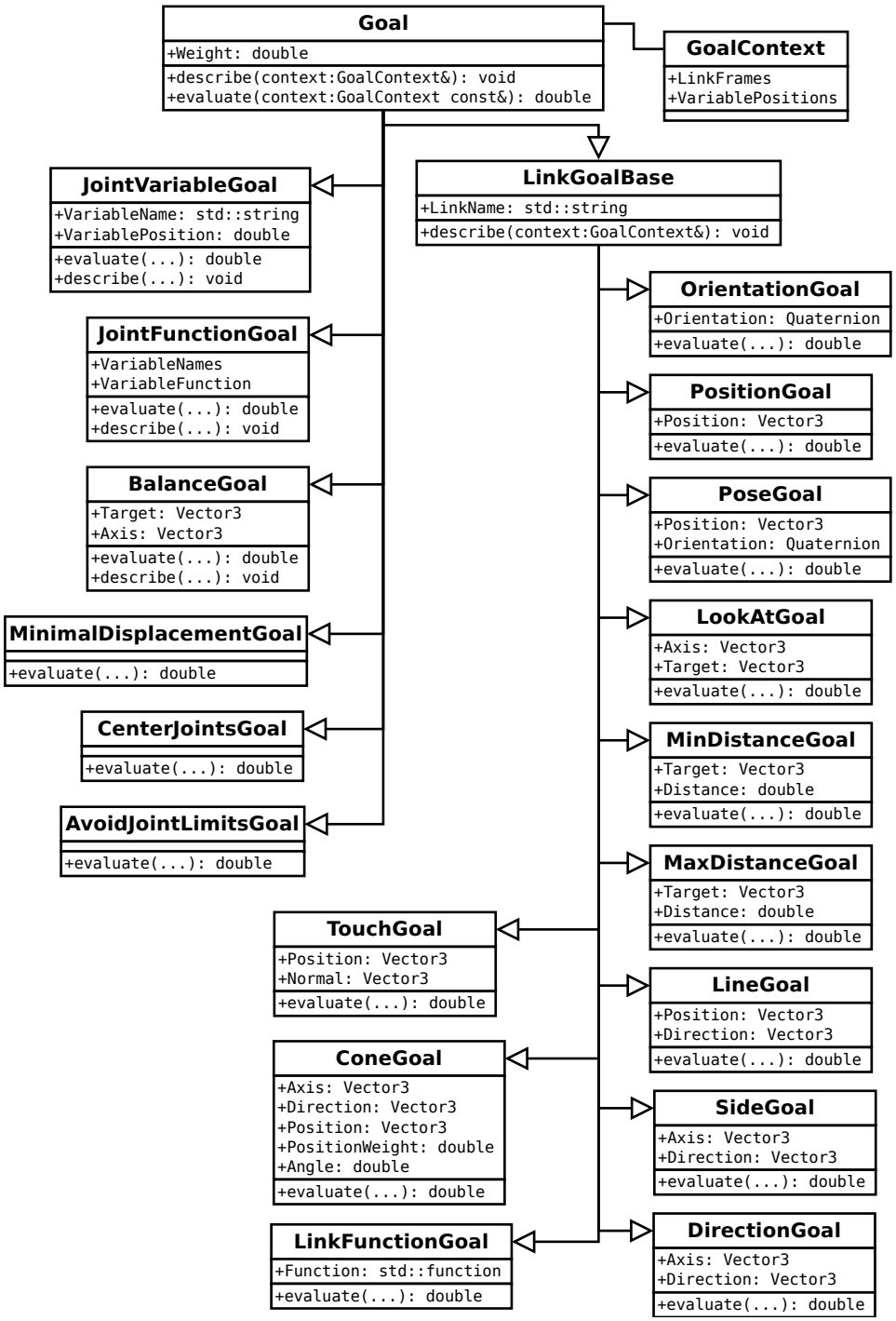

Figure 3.4: Goal class hierarchy

3.9.2 Inverse Kinematics Interface

The inverse kinematics plugin interface in MoveIt originally only accepted a list of goal poses, assuming all goals to be pose goals. The goal poses are stored by-value in a list, which prevents them from being extended for other goal types. The interface is implemented by BioIK for compatibility, but can only be used to specify pose or position goals. In addition to the goal poses, the MoveIt inverse kinematics plugin interface also accepts a $KinematicsQueryOptions$ parameter. This parameter is extended by a $BioIKKinematicsQueryOptions$ class to pass BioIK specific information to the solver. A *goals* parameter specifies a list of BioIK goals. It can be used for all existing BioIK goal classes as well as for new user-defined goal types. The goal pose list accepted by the IK methods can be disabled by setting a *replace* option in the $BioIKKinematicsQueryOptions$. A new $fixed\_joints$ parameter can be used to prevent a list of joints from being mutated by BioIK. The final fitness of the best found BioIK solution is returned in a $solution\_fitness$ field.

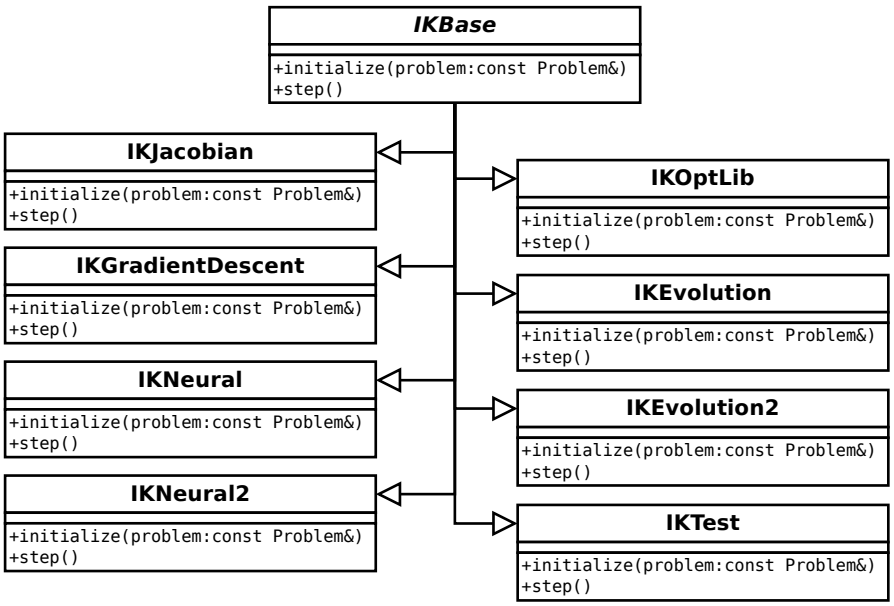

Figure 3.5: Solver classes

3.9.3 Solver Types

During research, several different solvers have been implemented (different variants of the evolutionary algorithm, gradient based methods for comparison, etc.). An internal abstraction has been implemented for different BioIK solvers. This allows faster iteration time, since new solvers do not have to be declared as ROS plugins, and it prevents the workspace from being needlessly cluttered (e.g. otherwise the MoveIt setup dialog would be listing a dozen different experimental implementations). The internal abstraction also provides a simpler interface, allowing easy experimentation with new IK methods. The implementation can be selected by setting a *mode* parameter on the ROS parameter server under the robot's kinematics namespace.

3.9.4 Multithreading

BioIK supports multithreading. During initialization, solver threads are started and paused using a thread barrier. When an inverse kinematics query arrives, the threads are resumed by triggering the barrier. A $finished$ flag is used to signal completion, upon which all threads deliver their best results and enter the barrier again. Data structures which are concurrently modified are copied once for each thread to prevent data races without needing additional synchronization.

3.9.5 Vectorization

Forward Kinematics

Most of the computation time is usually spent on forward kinematics extrapolations. These computations have been manually vectorized for different vector instruction sets (SSE2, AVX, FMA) using SIMD intrinsics. A non-vectorized version is retained compatibility. Function multiversioning is used to automatically select an appropriate implementation for the current CPU model when BioIK is loaded.

Alignment

For SIMD load and store operations, the aligned load and store instructions are used for optimal performance. By default, the GCC runtime library aligns memory allocations at 128bit boundaries. However, AVX has 256bit wide vector registers. Data which is accessed by AVX SIMD code is therefore explicitly aligned at 256bit boundaries.

Mutation

Reproduction and mutation have been optimized using auto vectorization. Manual vectorization has been avoided to not obfuscate the evolutionary algorithm. The reproduction and mutation function iterates over gene arrays stored in consecutive

memory and is thus a good candidate for auto vectorization. Code generation has been verified by disassembling the function using GDB. Auto vectorization and aligned memory access is requested using the OpenMP #pragma simd directive.

3.9.6 Source Tree

Public headers

`/include/bio_ik/frame.h` defines a fast coordinate frame class, which consists of a position vector and a rotation quaternion. It is used throughout BioIK for coordinate transformations.

`/include/bio_ik/goal.h` defines the `Goal` base class, a `GoalContext` class for communication between goal classes and solvers, and a `BioIKKinematicsQueryOptions` structure for passing BioIK specific information through the MoveIt interface.

`/include/bio_ik/goal_types.h` contains the different goal types. The IK solvers only include `goal.h`, but not `goal_types.h`. This allows the goal types to be changed without having to re-compile the solvers.

`/include/bio_ik/robot_info.h` defines a robot information class which stores joint limit data and is used by some of the goal types.

`/include/bio_ik/bio_ik.h` includes the other BioIK headers. Other packages should simply include this header.

Internals

`/src/goal_types.cpp` defines some of the goal class methods. Most goal methods are directly defined in the `goal_types.h` header, but some of the methods use specific library functions (e.g. for collision detection), so if they would be defined in the public header, and the public header would be included in another package, that other package would also have to explicitly link to the same libraries. This is avoided by defining some methods in `goal_types.cpp`, so that indirect dependencies are resolved implicitly through ROS package dependencies and the operating system's dynamic library loader. Defining the most complex goal methods separately also helps against code bloat and reduces compilation time.

`/src/forward_kinematics.h` contains classes for computing and extrapolating forward kinematics.

`/src/ik_base.h` contains a base class for the different optimization methods implemented in this work.

`/src/ik_parallel.h` implements multi-threading.

`/src/kinematics_plugin.cpp` contains the BioIK implementation of the MoveIt `kinematics::KinematicsBase` plugin interface.

`/src/problem.h` contains an IK problem description class, which manages goal lists and cached information about each goal, as well as additional information about the IK problem such as the initial guess and a list of active variables.

`/src/problem.cpp` implements methods related to IK problems; mainly initialization and determining active variables as well as end effector frames.

`/src/utils.h` defines internal utility classes and functions. This includes mathematical helper functions, a profiler which was used during optimization, a factory template for managing the different optimization methods, and an aligned vector class.

`/src/ik_evolution_1.cpp` implements the original BioIK algorithm, as well as minor optional optimizations.

`/src/ik_evolution_2.cpp` contains the re-designed BioIK algorithm.

`/src/ik_gradient.cpp` implements a custom multi-goal pseudo-inverse Jacobian solver as well as custom gradient descent methods for comparison.

`/src/ik_neural.cpp` implements neural network based IK methods.

`/src/ik_cppoptlib.cpp` implements IK solvers using different optimization methods provided by the CppNumericalSolvers library for comparison.

`/src/ik_test.cpp` defines a dummy IK solver, which—instead of solving IK problems—implements a self-test for the forward kinematics extrapolator.

3.10 Configuration

MoveIt can be configured to use BioIK using the MoveIt Setup Assistant.

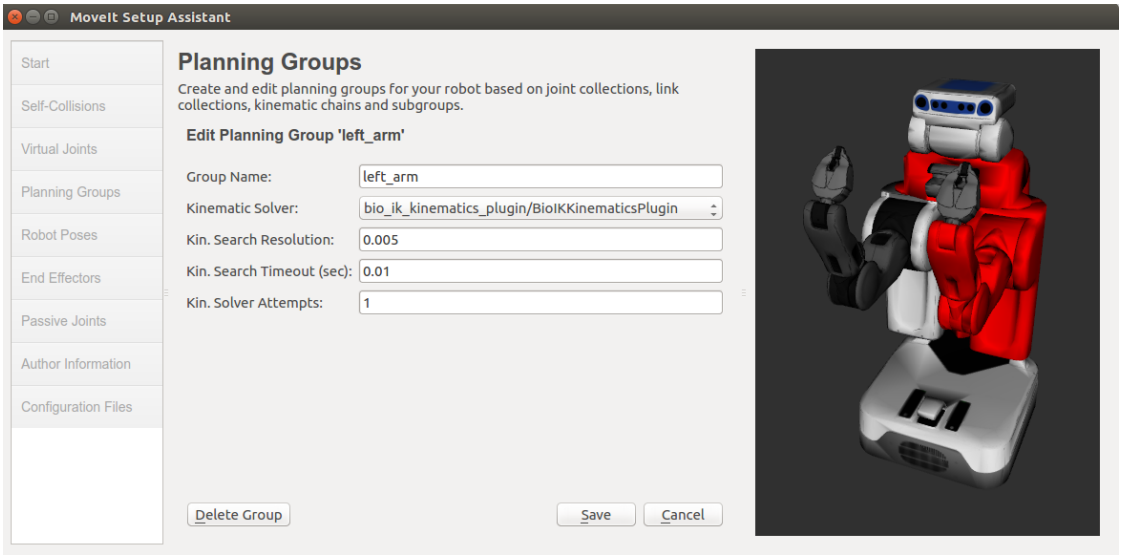

Figure 3.6: BioIK selected in the MoveIt Setup Assistant

BioIK can also be selected by editing the `config/kinematics.yaml` configuration file and manually setting the `kinematics_solver` property to `bio_ik_kinematics_plugin/BioIKKinematicsPlugin`.

Listing 3.1: kinematics.yaml manually edited for using BioIK on the PR2 robot

```
left_arm:

  # kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
  kinematics_solver: bio_ik_kinematics_plugin/BioIKKinematicsPlugin

  kinematics_solver_search_resolution: 0.005
  kinematics_solver_timeout: 0.005
  kinematics_solver_attempts: 1
```

The `kinematics.yaml` file is typically loaded by a launch file onto the ROS parameter server. So instead of editing the configuration file, it would also be possible to set the parameter on the parameter directly (e.g. using `rosparam set ...`).

Finally, the BioIK implementation of the `kinematics::KinematicsBase` MoveIt interface can also be instantiated programmatically via the ROS plugin interface or using the `kinematics_plugin_loader::KinematicsPluginLoader` class provided by MoveIt.

Chapter 4

Experiments

4.1 Forward Kinematics - Inverse Kinematics

Forward kinematics is used to generate goal poses that are guaranteed to be reachable. All joints are set to random joint values, forward kinematics is computed, and the end effector poses are used as the goal poses. The inverse kinematics solver is then called for the generated goal poses and the success rate as well as the average solve time is recorded.

	PR2	UR5	Valkyrie arm	Valkyrie foot	iiwa
BioIK 2	100.00%	99.93%	99.93%	100.00%	99.93%
BioIK 1	75.69%	51.20%	29.13%	70.14%	66.84%
TRAC_IK	99.91%	99.29%	99.64%	99.98%	99.88%
KDL	53.10%	41.70%	41.13%	91.68%	51.62%

Table 4.1: FK/IK benchmark: success rate

	PR2	UR5	Valkyrie arm	Valkyrie foot	iiwa
BioIK 2	0.45ms	0.50ms	0.51ms	0.28ms	0.47ms
TRAC_IK	0.77ms	0.52ms	0.73ms	0.18ms	0.41ms
BioIK 1	3.93ms	4.12ms	4.67ms	3.14ms	3.49ms
KDL	4.68ms	4.59ms	3.50ms	0.91ms	3.17ms

Table 4.2: FK/IK benchmark: average solve time

4.2 Grid Test

Goal poses are generated on a regular grid and the inverse kinematics solver is called for each generated goal pose. The timeout is set to 5ms, and the maximum error to 10^{-5}. The TRAC_IK and the re-designed BioIK algorithm both achieved 100% success rate on the PR2 and on the UR5 robot.

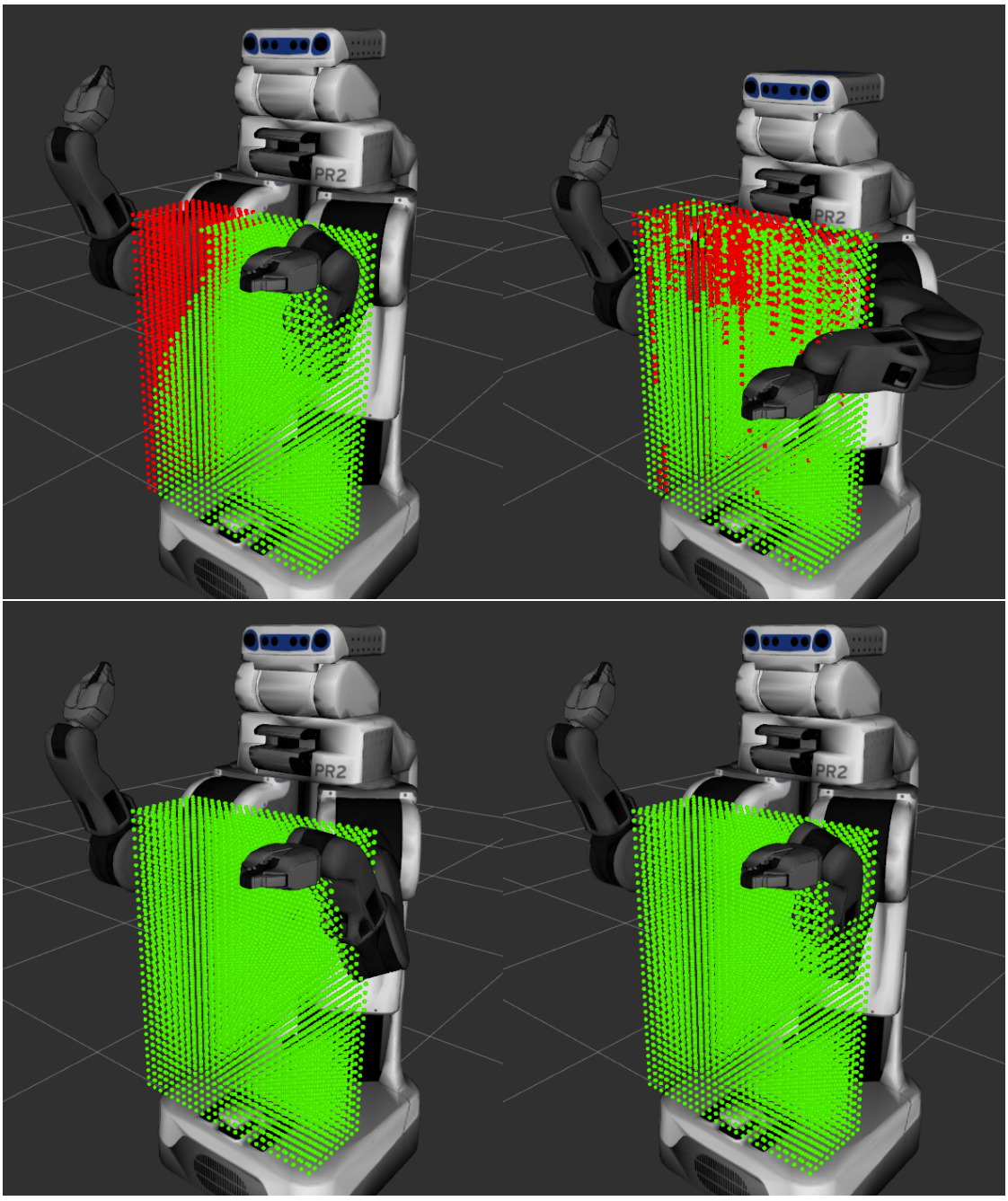

Figure 4.1: PR2 grid test - KDL (top left), BioIK 1 (top right), TRAC_IK (bottom left), BioIK 2 (bottom right)

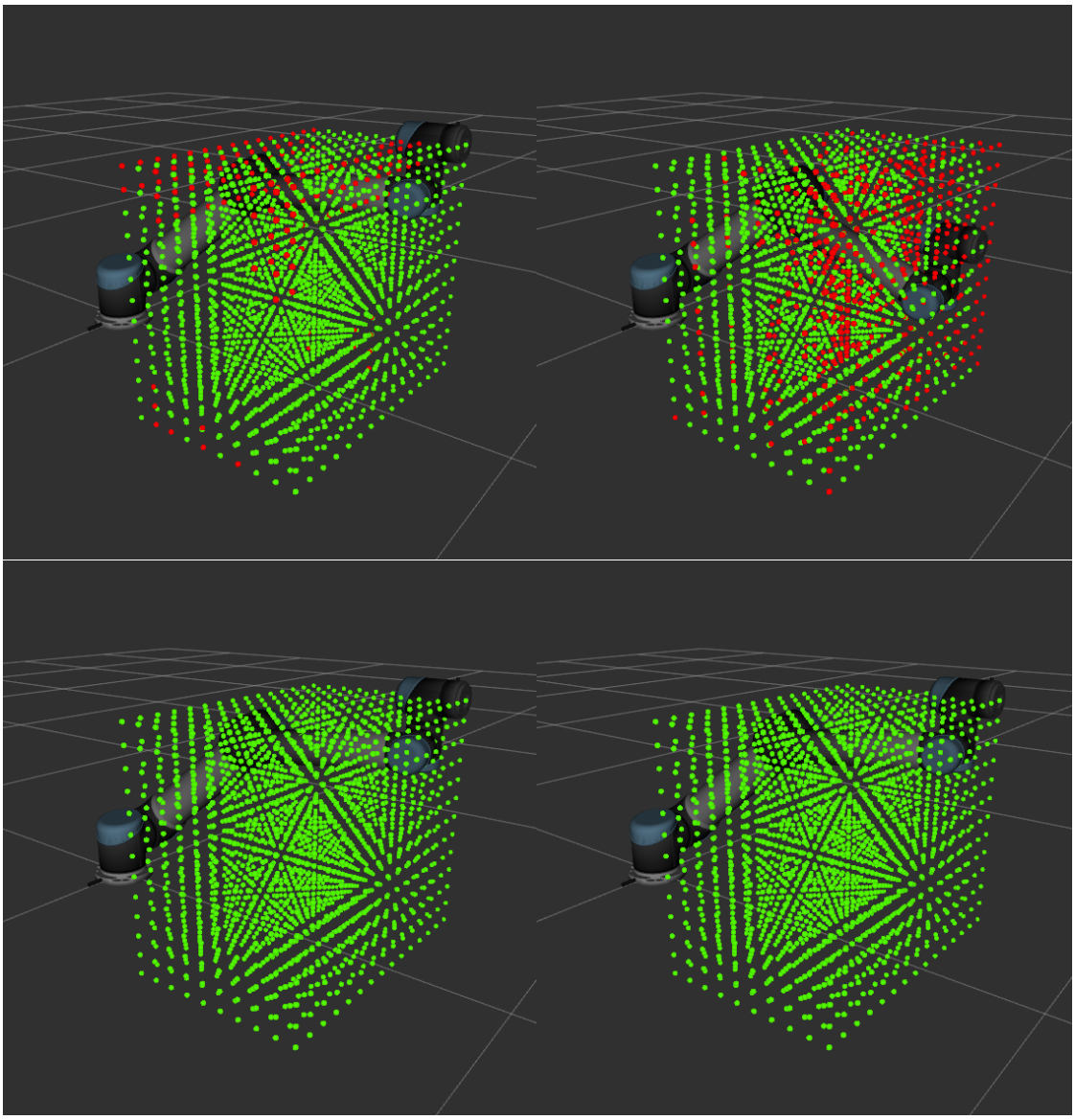

Figure 4.2: UR5 grid test - KDL (top left), BioIK 1 (top right), TRAC_IK (bottom left), BioIK 2 (bottom right)

4.3 Minimal Displacement

The PR2 robot can move it's upper body up and down using a vertical prismatic joint. However, the prismatic joint is relatively slow compared to e.g. the revolute joints in the arms. For many applications, good IK solutions should therefore try to avoid moving the vertical prismatic joint and instead prefer arm movements. This can be achieved using BioIK via a MinimalDisplacementGoal, which respects the maximum joint velocities. If possible, only the arms are movement. The relatively slow vertical prismatic joint is only moved if necessary. The default MoveIt IK solvers would either generate large vertical prismatic joint movements for most IK queries, even if not necessary, or the vertical prismatic joint would have to be manually disabled, in which case the workspace of the robot would be restricted and many goals which would be reachable otherwise could not be reached anymore.

In many cases, the KDL solver mainly moves the vertical prismatic joint.

With a MinimalDisplacementGoal, BioIK minimizes movement in the relatively slow vertical prismatic joint and mainly moves the arms, and only moves the vertical prismatic joint if necessary.

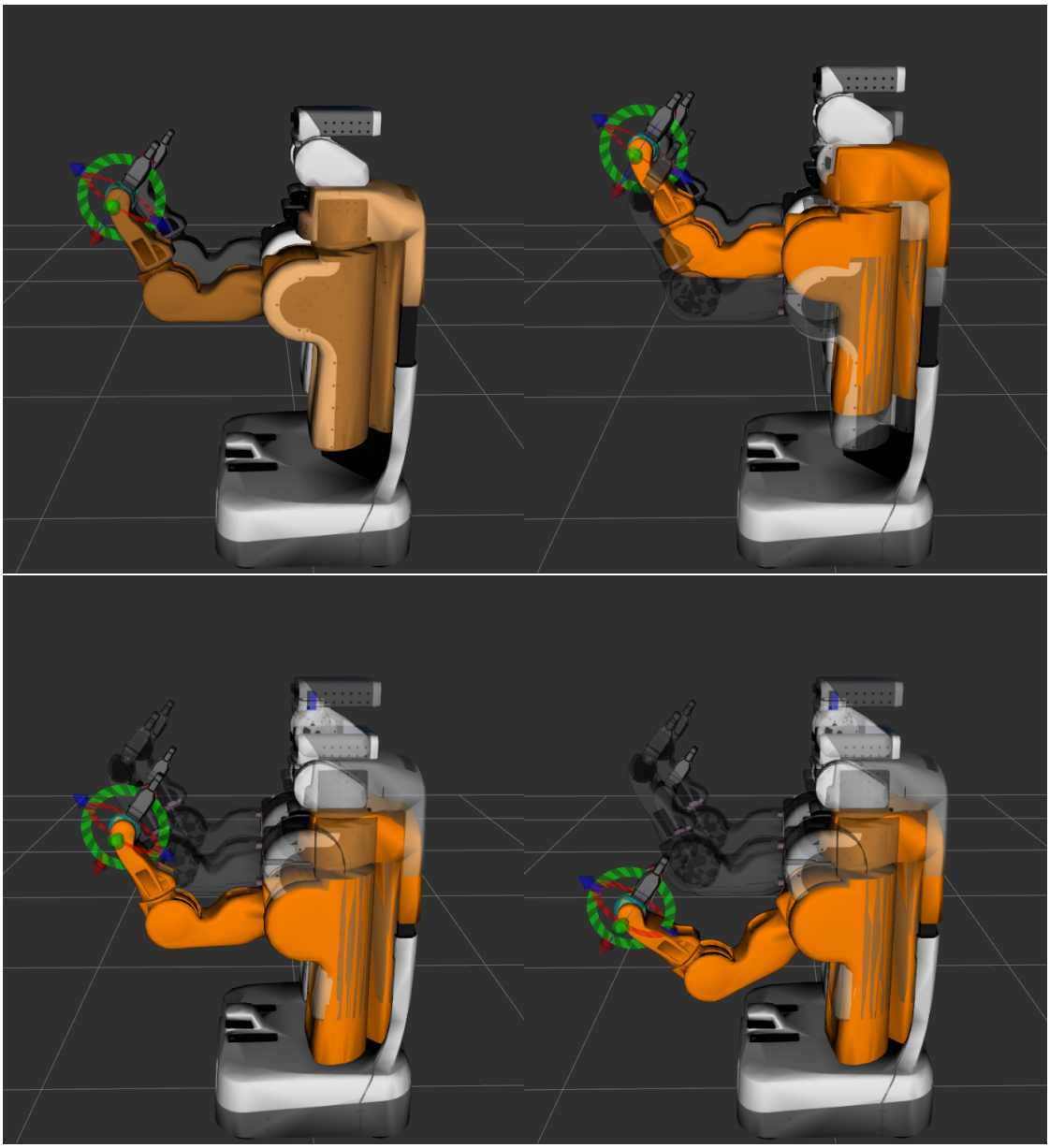

Figure 4.3: PR2 vertical prismatic joint movement - KDL

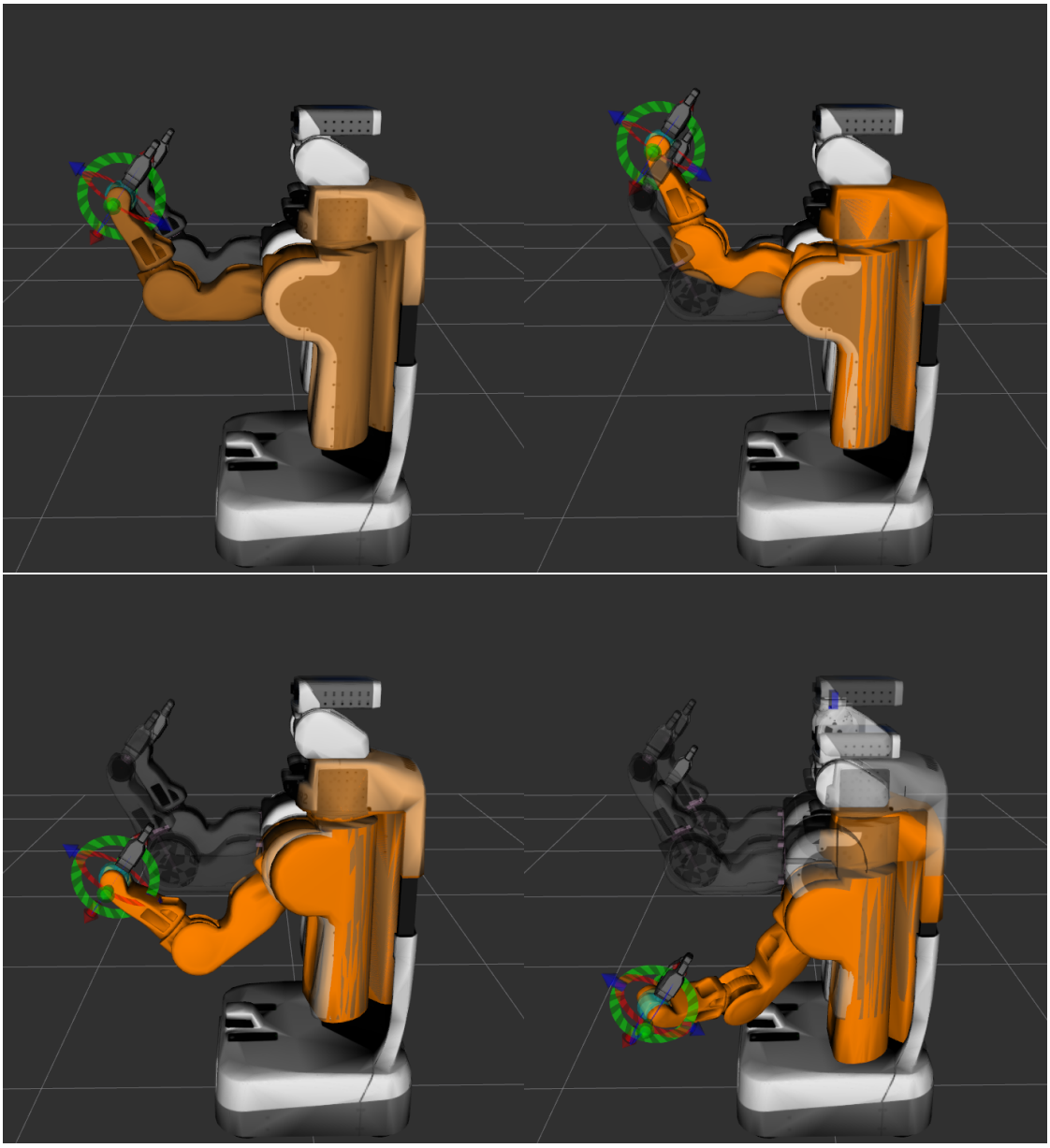

Figure 4.4: PR2 vertical prismatic joint movement - BioIK with MinimalDisplacementGoal

4.4 Valve Turning

A valve turning motion is generated for the PR2 robot through multi-goal inverse kinematics. One position goal is attached to each finger tip. The goals are moved on a circular path in front of the robot. The arms are controlled implicitly through multi-goal. IK goals are only defined for the fingers, but not for the arms or hands.

While turning in one direction, the finger tip goals at each hand are placed at a small distance from each other to grab the object. While rotating back in the other direction, the object is released by placing the finger tip goals at a larger distance from each other.

BioIK generates suitable arm and body poses to satisfy the position goals at the finger tips. A MinimalDisplacementGoal and a AvoidJointLimitsGoal are used to generate smooth motions.

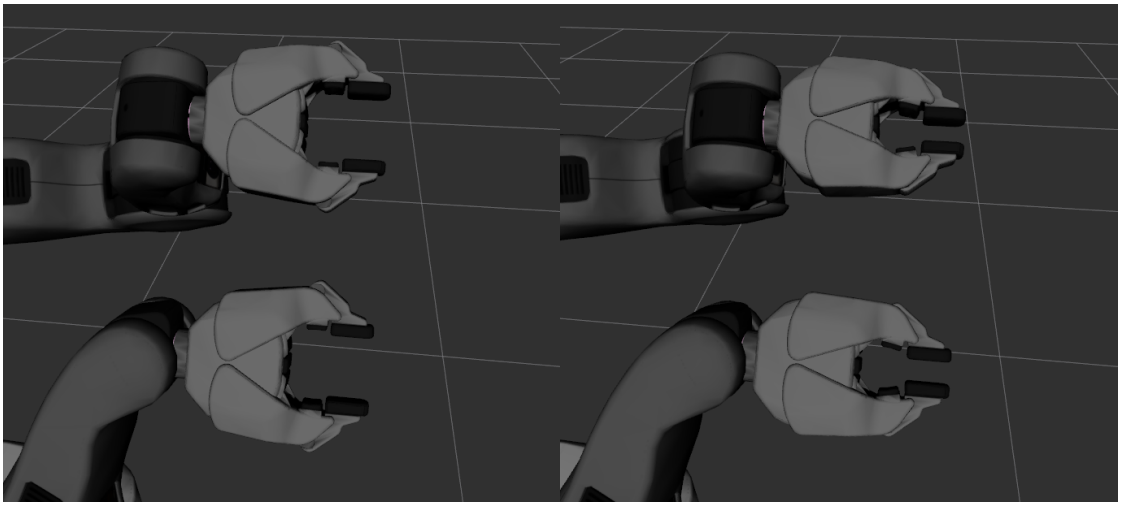

Figure 4.5: PR2 valve turning test - finger motions

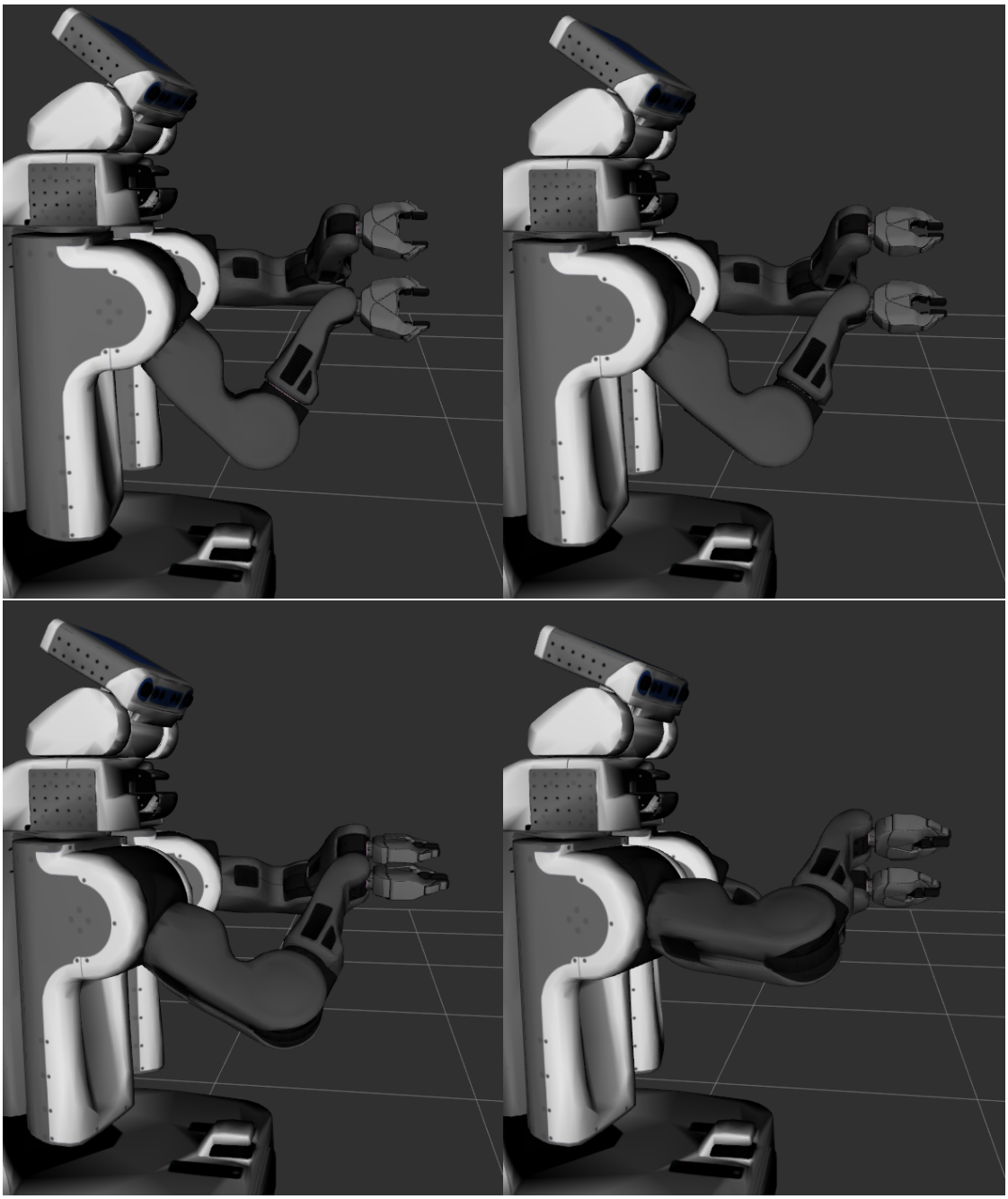

Figure 4.6: PR2 valve turning test - arm motions

4.5 Balanced IK

The BioIK solver is used to compute gravitationally balanced IK solutions for the NASA Valkyrie humanoid robot. The solutions are tested using the Gazebo simulator. Position goals are placed at the robot's wrists. A BalanceGoal is used to keep the robot's center of gravity above the feet. The BalanceGoal evaluates all links and their centers of gravity and affects all joints. Additional goals are added to pelvis, torso and head. Otherwise, solutions might be generated, which would require too large joint efforts. In principle, it should be possible to use the BioIK solver to automatically minimize joint efforts, but this has not yet been implemented. The additional goals also help with correctly initializing the simulation (Gazebo spawns the robot with the pelvis joint being upright). Additional foot reflexes have been added to the feet using a simulated IMU (inertial measurement unit) and a PD controller to counter dynamic forces which result from the inertia of the robot links during acceleration and deceleration, as well as outside forces, simulation instabilities, and joint controller inaccuracies. The position goals at the wrists can be moved interactively through interactive markers in RViz and the robot arms follow the position goals. The BalanceGoal automatically generates body movements which counter the arm movements to keep the robot's center of gravity above the feet. The simulated robot in the Gazebo simulation is able to keep standing without falling over. If the BalanceGoal is removed, the robot loses balance and falls over when the position goals are moved.

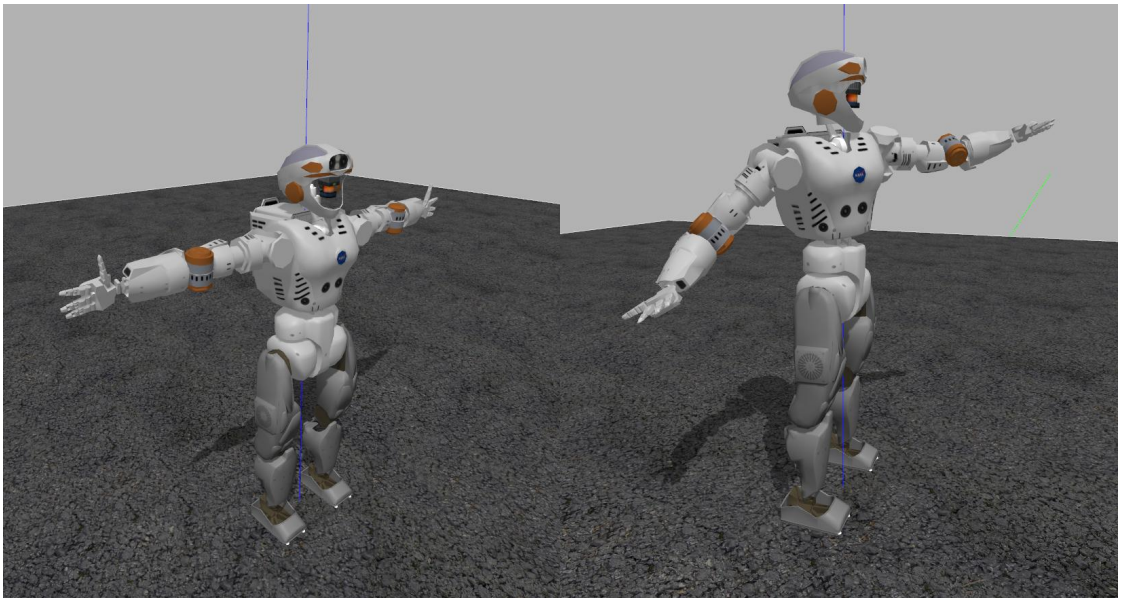

Figure 4.7: Balanced IK test - Gazebo simulation

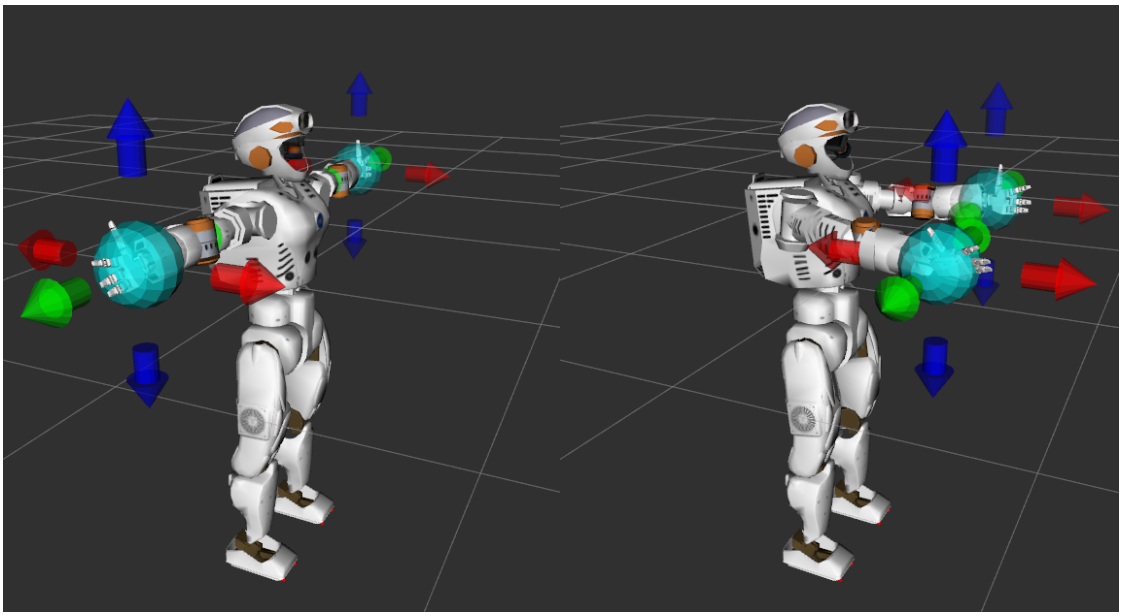

Figure 4.8: Balanced IK test - RViz controls

4.6 Shadow Hand

A robot setup with a C5 Shadow Dexterous Hand (humanoid robot hand) mounted to a KUKA LWR industrial robot arm is used at the TAMS research group.

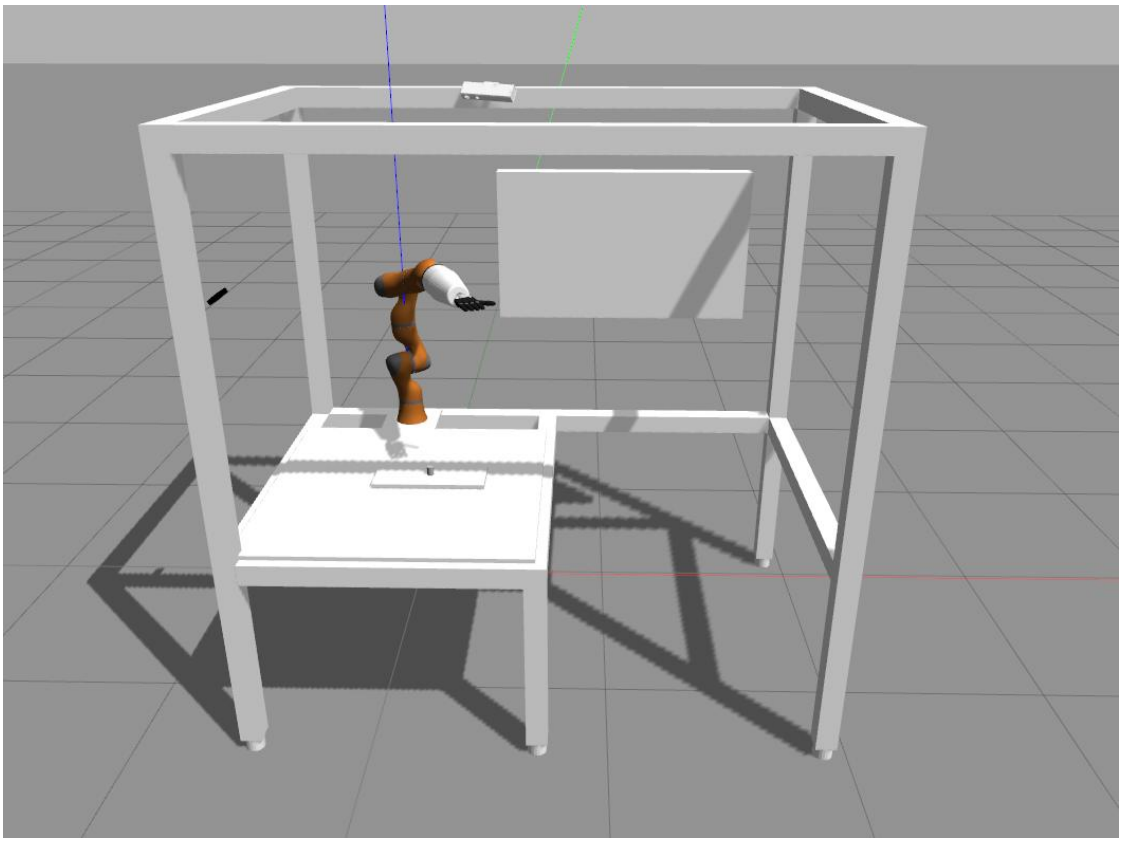

Figure 4.9: Shadow Hand setup

The BioIK solver is used to plan hand and arm motions for turning a wheel on an audio mixer. Each of the first three fingers is controlled via a TouchGoal, a LineGoal, and a LookAtGoal.

The TouchGoal is used to place the origin of the finger tip frame at the correct distance from the object for the finger to touch the object. The position of the finger tip could also be controlled via a simple position goal. In this case, an offset away from the surface would have to be added manually to account for the thickness of the finger. Since the finger tip is not a perfect sphere, the ideal finger tip offset depends on the position and orientation of the finger, and a constant offset would lead to inaccurate results. If a TouchGoal is used, a manually configured fixed offset is not needed, and the position and orientation dependent ideal offset is computed automatically and accurately from the finger's collision model.

The LineGoal points out of the object and is used to control the contact point. BioIK keeps the position of the finger tip link on the line defined by the LineGoal. The contact point can be rotated around the object by rotating the LineGoal.

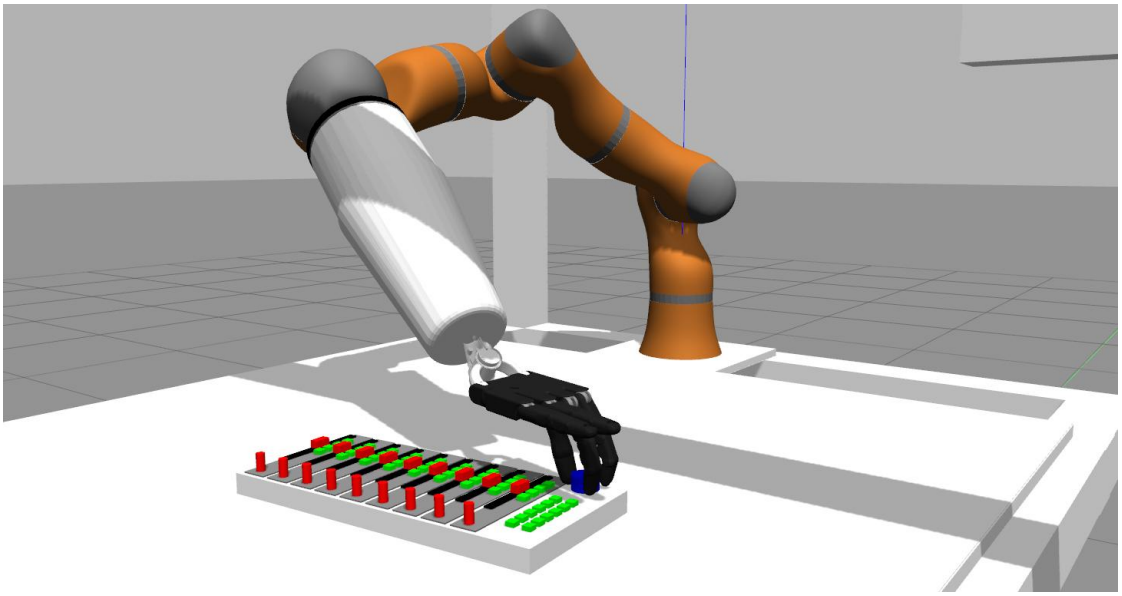

Figure 4.10: Shadow Hand experiment - Gazebo simulation

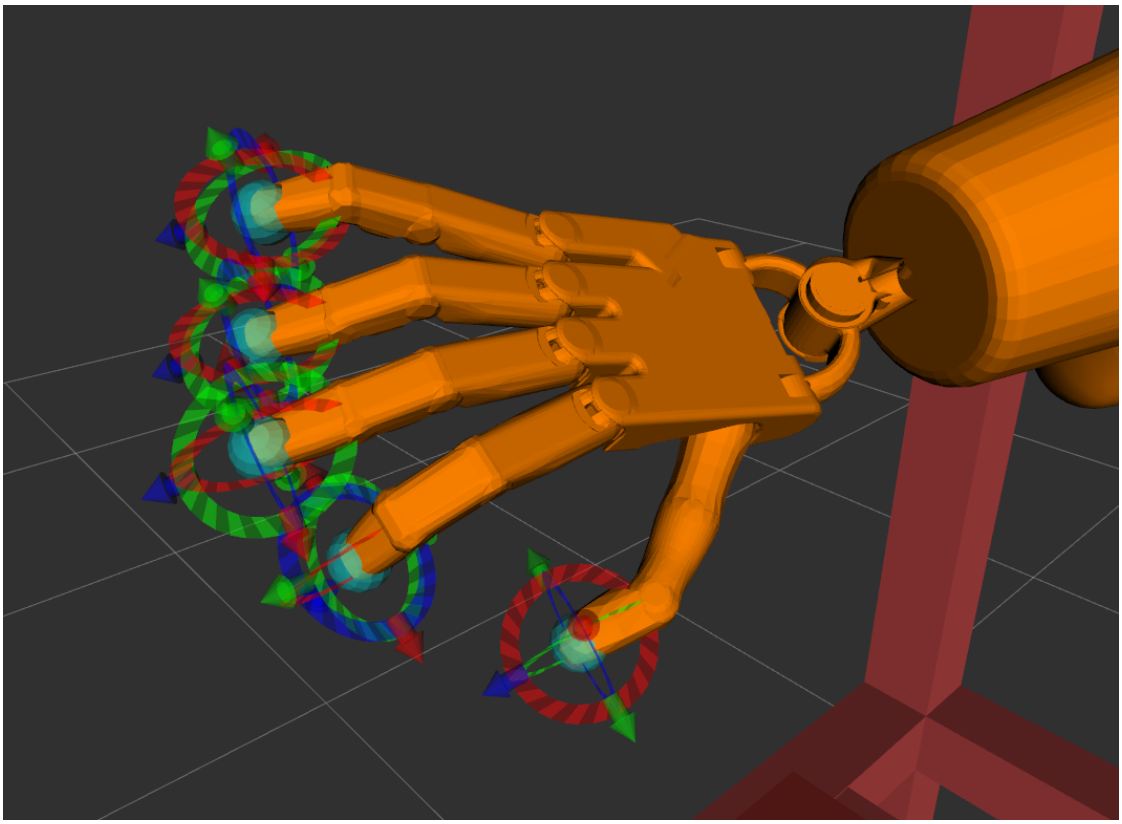

Figure 4.11: Shadow Hand with one position goal at each finger tip

The LookAt goal is used to make sure that the object is touched with the correct side of the finger.

The motion is generated by rotating the finger tip goals around the object. Before rotating the object, the finger tips start at a small outward offset and slightly above the object. The goals are then lowered and closed to grab the object. A joint trajectory is generated by computing a sequence of 200 inverse kinematics solutions and combining them into a trajectory.

A MinimalDisplacementGoal is used to prefer small joint movements from one step to another and to generate a smooth trajectory.

A CenterJointsGoal keeps solutions away from joint limits. Reaching the joint limits could force solutions to be generated that would result in velocity discontinuities. These could cause the joint controllers and joint motors to be unable to accurately follow the trajectory and the fingers would slip off. A trajectory with velocity discontinuities could be transformed into a trajectory which would be executed accurately by adjusting the timing. But then, the wheel could not be turned at a constant speed anymore.

The MinimalDisplacementGoal and the CenterJointsGoal are specified as secondary objectives. If defined as primary objectives, the results would be skewed towards the center of each joint's operating range and towards previous solutions, and the finger tips would not accurately follow the finger tip goals.

Since a human arm is attached to the side of the body, it has an intrinsically preferred direction for accessing positions withing the workspace. In the robot setup used for this experiment, the arm is mounted vertically on a table. JointVariableGoals assigned to the first two robot links and defined as secondary objectives are used to prefer one direction and to generate movements that roughly match the behavior of the right arm of a human.

Before executing the trajectory, the robot is moved to the start pose by calling a standard MoveIt motion planner.

The generated trajectories are tested in a Gazebo simulation. The robot is able to smoothly turn the wheel without slipping off.

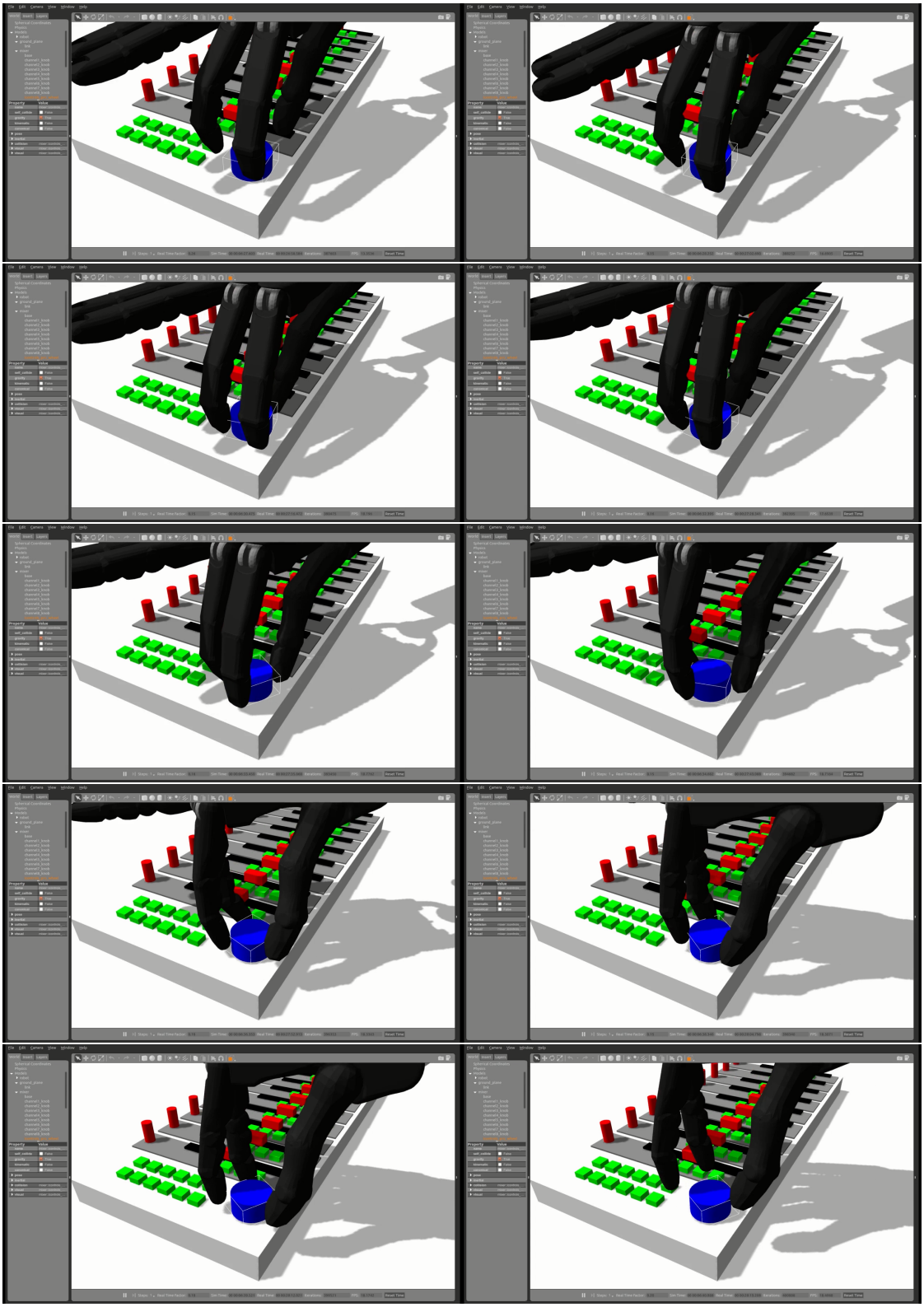

Figure 4.12: Shadow Hand experiment - Gazebo simulation

Chapter 5

Conclusion

In this work, an evolutionary multi-goal inverse kinematics solver has been implemented for MoveIt and ROS. Performance has been optimized through algorithmic improvements and efficient implementation. In many cases, the newly developed method outperforms existing gradient-based methods, including single-goal inverse kinematics solvers which were already available in MoveIt, as well as other multi-goal solvers which have been implemented in this work for comparison. The implementation has been tested through benchmarks and experiments. Several different IK goal types are provided, many of which were not previously available in MoveIt. The functionality can be extended by adding new goal classes and implementing cost functions.

Bibliography

[1] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[2] Robot Operating System. `http://www.ros.org/`. Accessed: 2017-07-21.

[3] ROS packages. `http://www.ros.org/browse/list.php`. Accessed: 2017-07-21.

[4] MoveIt. `http://moveit.ros.org/`. Accessed: 2017-07-21.

[5] Sachin Chitta, Ioan Alexandru Sucan, and Steve Cousins. Moveit! [ros topics]. *IEEE Robot. Automat. Mag.*, 19:18–19, 2012.

[6] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[7] Gazebo. `http://gazebosim.org/`. Accessed: 2017-07-21.

[8] Orocos Kinematics and Dynamics Library. `http://www.orocos.org/kdl`. Accessed: 2017-07-21.

[9] Patrick Beeson and Barrett Ames. TRAC-IK: An open-source library for improved solving of generic inverse kinematics. In *Proceedings of the IEEE RAS Humanoids Conference*, Seoul, Korea, November 2015.

[10] TRAC-IK. `https://bitbucket.org/traclabs/trac_ik.git`. Accessed: 2017-07-21.

[11] Open Motion Planning Library. `http://ompl.kavrakilab.org/`. Accessed: 2017-07-21.

[12] Intel Intrinsics Guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. Accessed: 2017-07-21.

[13] GCC Function Multiversioning. `https://gcc.gnu.org/onlinedocs/gcc/Function-Multiversioning.html`. Accessed: 2017-07-21.

[14] PR2 Robot. `http://wiki.ros.org/pr2_description`. Accessed: 2017-07-21.

[15] UR5 Robot. `http://wiki.ros.org/ur5_description`. Accessed: 2017-07-21.

[16] Valkyrie Robot Wiki. `https://github.com/NASA-JSC-Robotics/valkyrie`. Accessed: 2017-07-21.

[17] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[18] Flexible Collision Library. `https://github.com/flexible-collision-library/fcl`. Accessed: 2017-07-21.

[19] S. Starke. *A Hybrid Genetic Swarm Algorithm for Interactive Inverse Kinematics*. 2016.

[20] Sebastian Starke, Norman Hendrich, Sven Magg, and Jianwei Zhang. An efficient hybridization of genetic algorithms and particle swarm optimization for inverse kinematics. In *2016 IEEE International Conference on Robotics and Biomimetics, ROBIO 2016, Qingdao, China, December 3-7, 2016* [21], pages 1782–1789.

[21] *2016 IEEE International Conference on Robotics and Biomimetics, ROBIO 2016, Qingdao, China, December 3-7, 2016*. IEEE, 2016.

[22] Sebastian Starke, Norman Hendrich, and Jianwei Zhang. A memetic evolutionary algorithm for real-time articulated kinematic motion. 2017.

[23] C. Natale. *Interaction Control of Robot Manipulators: Six degrees-of-freedom tasks*. Springer Tracts in Advanced Robotics. Springer Berlin Heidelberg, 2003.

[24] B Durmuş, H Temurtaş, and A Gün. An inverse kinematics solution using particle swarm optimization. In *International Advanced Technologies Symposium (IATS'11)*, pages 16–18, 2011.

[25] Samuel R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. 2004.

[26] Thomas Collins and Wei-Min Shen. Paso: An integrated, scalable pso-based optimization framework for hyper-redundant manipulator path planning and inverse kinematics. 2014.

[27] Omar Alejandro Aguilar and Joel Carlos Huegel. Inverse kinematics solution for robotic manipulators using a cuda-based parallel genetic algorithm. In Ildar Z. Batyrshin and Grigori Sidorov, editors, *MICAI (1)*, volume 7094 of *Lecture Notes in Computer Science*, pages 490–503. Springer, 2011.

[28] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[29] S. Nissen. Implementation of a fast artificial neural network library (fann). Technical report, Department of Computer Science University of Copenhagen (DIKU), 2003. http://fann.sf.net.

[30] B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer Handbooks. Springer International Publishing, 2016.

[31] F. Dunn and I. Parberry. *3D Math Primer for Graphics and Game Development*. Wordware game math library. Wordware Pub., 2002.

[32] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, April 2013.

[33] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[34] Shadow Robot. `https://github.com/shadow-robot`. Accessed: 2017-07-21.

[35] KUKA iiwa. `https://www.kuka.com/en-us/products/robotics-systems/industrial-robots/lbr-iiwa`. Accessed: 2017-07-21.

[36] Unity3D. `https://unity3d.com`. Accessed: 2017-07-21.

[37] C++ Reference. `http://en.cppreference.com/w/`. Accessed: 2017-07-21.

[38] CppNumericalSolvers. `https://github.com/PatWie/CppNumericalSolvers`. Accessed: 2017-07-21.

[39] Agner Fog. Instruction Tables. `http://www.agner.org/optimize/instruction_tables.pdf`. Accessed: 2017-07-21.

Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Masterthesis im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum Unterschrift

Erklärung zur Veröffentlichung

Ich stimme der Einstellung der Masterthesis in die Bibliothek des Fachbereichs Informatik zu.

Ort, Datum Unterschrift