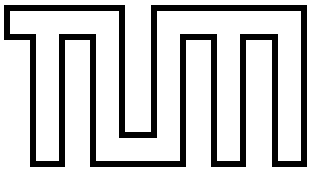


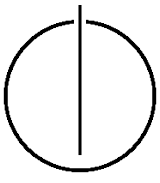
FAKULTÄT FÜR INFORMATIK

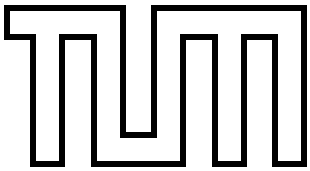
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's thesis
Robotics, Cognition, Intelligence

Realizing Online (Self-)Collision Avoidance Based on Inequality Constraints with Hierarchical Inverse Kinematics

Karsten Knese





FAKULTÄT FÜR INFORMATIK

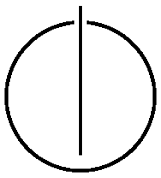
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's thesis
Robotics, Cognition, Intelligence

Realizing Online (Self-)Collision Avoidance Based on Inequality Constraints with Hierarchical Inverse Kinematics

Realisierung von Selbstkollisionsvermeidungen auf Basis von Ungleichheitsbedingungen mit Hierarchisch Inverser Kinematik

Author:	Karsten Knese
Supervisor:	Prof. Dr. Daniel Cremers
Advisor:	Dr. Jürgen Sturm (Technische Universität München)
	Adolfo Rodriguez Tsouroukdissian, PhD (PAL Robotics S.L)
Date:	April 14, 2014



Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this Master's thesis, only supported by declared resources.

München, den 14.4.2014 Karsten Knese

Acknowledgments

So here it is, my Master's thesis! Despite the fact, that my name is stated as the single author of this work, this document is written in a 'we' formulation. And this has to be taken seriously! This work could not have been realized in the way it is without the help and support of various talented minds.

To begin with, my highest appreciation and biggest 'thank you' belongs to Gennaro Raiola, for his elementary work of bringing the Stack-of-Tasks to life inside PAL Robotics. He was not only a patient colleague, but furthermore became a good friend, who made my time in Barcelona as good as it was.

I want to thank Adolfo Rodríguez Tsouroukdissian, who mentored my work inside PAL and guided me through all arising troubles with smart ideas. Equally to Jürgen Sturm, who took not just care about all academic bureaucracy, but also gave me valuable feedback for designing this work.

Furthermore, a big thanks goes to Hilario Tomé, who was patient and compassionated enough to reveal all necessary mathematical myths, I encountered during this work; Paul Mathieu and Bence Magyar for enlightening me with insights of C++ and ROS; Various lovely people all around the globe for cross-reading this document.

Altogether, I want to thank my colleagues at PAL Robotics, who are a great team and make working there as much fun as it could be.

Abstract

Humanoid robots comprise a high number of redundancy, which can be used to achieve multiple goals in parallel. A hierarchy of tasks is often implemented in order to execute them simultaneously. However, this emerges a high risk of (self-)collisions, where a reactive and online collision avoidance has to be instantiated inside the hierarchy.

In this Master's thesis, a solution for highly dynamic (self-)collision avoidance is presented. The collision avoidance is realized as an inequality constraint inside the Stack-of-Tasks.

Our proposed (self-)collision avoidance is implemented for the quadratic programming solver inside the Stack-of-Tasks. The alignment of linear constraints in form of a stack enables an iterative solving progress with respect to priorities given to these constraints. Hence the name, Stack-of-Tasks. The collision avoidance task is based on the closest point pair of possible collisions between two body parts. To achieve a unique solution for the closest point pair, a complete capsule decomposition for the robot model is created. The pseudo convexity of capsules reduces the number of discontinuities and leads to a smooth transition. The velocity along the unit-vector of these two points is limited by a threshold, dependent on the corresponding distance. The projection of the Jacobian towards the collision center point is reduced to zero if the inequality boundary is hit, thus disabling any movement towards the collision center. The same method can be used for self-collision avoidance as well as avoidance of external collision objects.

In order to keep the computational cost to a minimum, the dynamic graph inside the SoT is exploited in a way, that only the necessary closest point pairs are recomputed. This implies an efficient on-demand computation of the collision matrix. The self-collision is placed on a high level inside the hierarchy to ensure a constant collision check. On the same hand, the applied method allows a smooth execution of lower prioritized tasks of interest.

Abstract - Deutsch

Humanoide Roboter besitzen eine Vielzahl von redundanten Aktuatoren, die benutzt werden können, um mehrere Aufgaben parallel auszuführen. Dabei ist oft eine Hierarchie zwischen diesen Aufgaben realisiert. Das gleichzeitige Ausführen von Zielpositionen birgt jedoch die Gefahr von (Selbst-)Kollisionen. Das erfordert eine Kollisionsvermeidung in Echtzeit innerhalb dieser Hierarchie.

In dieser Masterarbeit präsentieren wir eine vollständige Lösung für eine hoch-dynamische (Selbst-)Kollisionsvermeidung. Diese ist realisiert als eine Ungleichheitsbedingung innerhalb des Stack-of-Tasks.

Unsere vorgeschlagene Methode zur (Selbst-)Kollisionsvermeidung ist implementiert für den Quadratic Program Solver innerhalb des Stack-of-Tasks. Die Anordnung der linearen Bedingungen geschieht in Form eines Stacks, was eine iterative Lösung mit Respekt zu den gegebenen Prioritäten erlaubt. Daher der Name Stack-of-Tasks. The Kollisionsvermeidung basiert auf der Berechnung der kürzesten Distanz zwischen zwei Kollisionsobjekten. Um eine eindeutige Lösung für diese Distanz zu erzielen, schlagen wir eine Darstellung des Roboterkollisionsmodells durch Kapseln vor. Die Pseudokonvenxität von Kapseln verringert die möglichen Diskontinuitäten. Die Geschwindigkeit entlang des Einheitsvektors der kürzesten Distanz is limitiert durch einen Schwellwert. Die Projektion der Jacobianmatrix auf den Richtungsvektor wird auf Null reduziert, wenn der Grenzwert der Ungleichheit erreicht ist. Die gleiche Methode kann für Selbstkollisions- als auch für externe Kollisionsvermeidung verwendet werden.

Um die Ausführungszeit minimal zu halten, wird der Dynamic Graph innerhalb des SoT aufgenutzt um nur die notwendigen kürzesten Distanzen zu berechnen. Die Kollisionsvermeidung besitzt eine hohe Priorität innerhalb der Hierarchie. Gleichzeitig erlaubt unsere Methode eine reibungsfreie Ausführung von niedrig priorisierten Aufgaben.

CONTENTS

Part I.

Introduction and Theory

CHAPTER 1

MOTIVATION

1.1. Problem Statement

Robot technology is constantly increasing as the performance of the used hardware components enable highly complex software computations. State-of-the-art robots such as REEM-C [51], Asimo [1], Hubo [19] and Atlas [8] indicate that humanoid robots are capable of achieving versatile goals in appropriate time. Videos such as this[1] expose the remarkable speed of kinematic capabilities. The dexterity and execution speed of humanoid

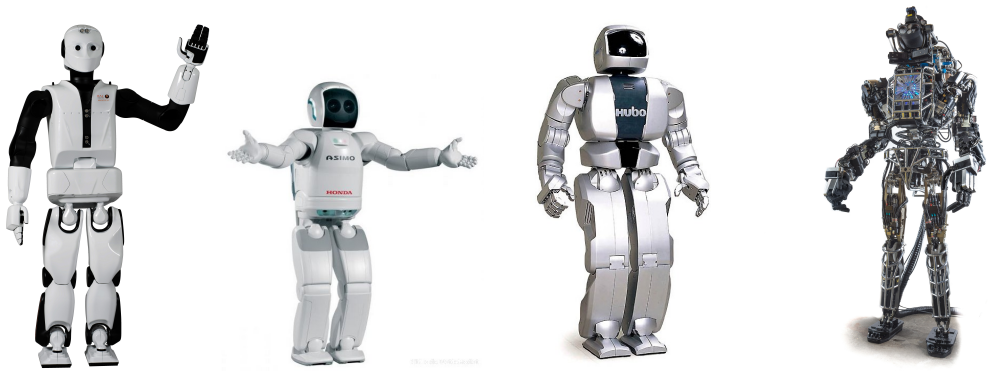


Figure 1.1.: State-of-the-art humanoid robots. From left: REEM-C, Asimo, Hubo, Atlas

robots thus encourage their integration into human environments. The idea of a robot accompanying a human in his daily life is becoming closer to practice. Complex tasks such as cleaning, ironing clothes or acting as a personal assistant are no longer futuristic visionary ideas, but already realizable at this time.

[1] http://www.youtube.com/watch?v=N_m56irWKeI

What follows is the necessity of addressing safety issues, which emerge as a consequence of high execution velocities. Collisions with the robot environment as well as with the robot themselves have to be stringently avoided at all times. This becomes non-trivial especially with versatile humanoid robots, as they might easily collide with themselves, due to the redundant degrees of freedom (DOF) or the overlapping workspace of the attached manipulators (hand to hand collision).

One way to handle collisions can be achieved by utilizing offline motion planners [31]. The complete scene is captured and an optimal solution without any collisions is planned. Once the trajectory is found, it is executed on the robot [34]. In order to increase the speed of this planning step, especially with respect to avoiding obstacles, various optimization approaches are suggested. Rapidly-Exploring Random Trees (RRT) [30] or Kinodynamic Motion Planning by Interior-Exterior Cell Exploration (KPIECE) [53] are two methods, which are widely integrated in current motion planning libraries [26]. Probabilistic Roadmaps (PRM) [25] are equally efficient, when the complete scene is known a priori. Probabilistic approaches [55] are generally used to find an optimal solution.

As the name *offline* planner suggests, the trajectory planning and collision avoidance occur before the actual execution. Since the perception of the environment is still computational expensive, it is not surprising that those approaches are less efficient in dynamic environments. A fast-changing environment demands a highly reactive change of the current trajectory, still avoiding any collisions.
Reactive collision avoidance algorithms try to overcome this issue. A first approach was based on repulsive forces emitted by the obstacles [27]. Real time enabled collision avoidance based on repulsive forces have been broadly used on humanoid robots [54] [59] [47]. An enhanced development of this approach has been examined by calculating the closest point pairs between two collision objects and orient the repulsive force along the resulting directional vector [9]. However, this implies a continuity of these calculated point pairs in order to avoid singularities and the resulting high velocities on the robot actuators [14].

Since humanoid robots are highly over-actuated systems, their redundancy implies a nullspace optimization [4], in which multiple goals can be achieved simultaneously. Within this, whole body motion control describes how to achieve a desired goal pose, utilizing all available joints. However, since multiple goal tasks can conflict with each other, priority levels are commonly used [16][35] and can be solved in a recursive hierarchical manner [50]. We have therefore developed a method that ensures a collision avoidance on a high priority, but equally provides a sufficiently large nullspace to execute lower priority tasks.

1.2. Goals of this thesis

This work presents a complete solution for external as well as self-collision avoidance of humanoid robots. The presented implementation is based on the proposed algorithm in [52]. The collision avoidance is realized as an inequality constraint inside the Stack-of-Tasks (SoT) [38], which serves as the base framework throughout this work. The SoT provides a highly efficient solver for hierarchical quadratic programming, which allows the execution of simultaneous goals in a whole body control manner. The algorithm in [52] restricts any movement along the directional vector between two closest points. In order to achieve a smooth trajectory of the closest point pair, we suggest a complete capsule decomposition as the robot's collision representation. The closest point pair is thus calculated between two capsules in order to ensure a smooth continuation in time.
This work is developed on two humanoid robots from PAL Robotics S.L., namely REEM-H and REEM-C. Since both robots are completely ROS based, the first goal is to integrate the SoT inside the ROS environment.

As a summary, solutions for the following goals are presented:

- Integrate the SoT inside a ROS based environment to enable the execution on PAL Robotics humanoid robots
- Assemble the robot collision model through a complete capsule decomposition to allow a smooth and continuous calculation of a closest point pair
- Implement a collision avoidance task inside the SoT, utilizing the calculated point pair to ensure no self-collision and to enable support for external collision avoidance.

1.3. Overview of this thesis

Part I of this work explains all the theoretical details, which are necessary for understanding the concepts implemented in this work. After this introduction, chapter 2 introduces the Stack-of-Tasks (SoT) from a theoretical point of view. We present background information about how to specify tasks based on equality and inequality constraints inside a quadratic program. Since the main objective of this thesis is the development of a collision avoidance task, we dedicate chapter 3 to details on the implemented method.

Implementation details and practical experiments, conducted on the robot are presented in part II. We briefly introduce the implementation environment in chapter 4, where we describe ROS_control as the main control framework, running on REEM-H and REEM-C. We further give information on how to integrate the SoT into a ROS based environment. Since the SoT comprises a complete framework, including the numerical solver, we describe the main functionalities, which are necessary for implementing a (self-)collision task. The

main method, which is implemented to achieve a successful collision avoidance, has to be supplemented with additional tasks, e.g. joint limits. The instantiated tasks, which run on the robot to perform successful collision avoidance, are presented in chapter 5. In order to evaluate the implemented methods, various tests for all specified goals of chapter 1.2 are examined and analyzed in chapter 6. Finally, in chapter 8 we provide applications, which are successfully realized and give an outlook about further developments.

CHAPTER 2

RELATED WORK - STACK OF TASKS

The Stack-of-Tasks (SoT) is an efficient framework for hierarchical quadratic programming (QP) [23]. It implements a General Inverse Kinematics (GIK), which was firstly presented in [60]. A task denotes an abstract desired goal value, which can be achieved by controlling an ordinary differential equation [16]. Tasks can be, among others, a desired pose of a manipulator, a gaze task describing a point in a 2-dimensional plane or a range of allowed joint values. Whole body motion control implies a high number of DoF and with it a lot of redundancy. Thus it becomes obvious that multiple instances of those tasks can be regulated simultaneously [33]. A widely common approach is to install a hierarchy of tasks, solving them recursively in respect of the precedent nullspace [50].

Generally, the SoT provides two main functionalities. Firstly, it incorporates a QP solver in a hierarchy preserving manner [37]. Hereby, a novel and highly computationally efficient matrix decomposition was introduced, which yields a remarkable improvement in terms of computation speed [12]. [11] refers to this specific development as Hierarchical Complete Orthogonal Decomposition (HCOD), which is adopted throughout this work when the emphasis is attached to the solver itself rather than the overall framework.

Secondly, the SoT describes a software framework, which implements an intelligent way of establishing a dynamic hierarchy by activating and deactivating task constraints on runtime. In the remainder of this chapter, the focus lies on the theoretical aspect of numerically solving the hierarchy of tasks. Complementary, the implementation of the software framework takes place in the second part of this work in chapter 4.

2.1. Prerequisites

The SoT is the base line of this work. In order to ease the understanding of the concepts presented in the following, a few requisites are recalled. The inverse kinematic solver mainly works with a least-square pseudo inverse approach, which can be formulated as a constrained quadratic programming problem. Finally, the SoT implements an inverse kinematics control scheme to run in real time on the robot.

2.1.1. Least Square

Least Squares is a way to optimize problems to its best. It applies always to overdetermined problems, where there exists no unique solution. This is the case, when there are more linearly independent equations than unknowns.

$$Ax = b \quad \text{with } A \in \mathbb{R}^{m \times n}, m < n \tag{2.1}$$

Since A is strictly skinny matrix, there is no direct inverse in order to analytically solve this equation system. However, we can optimize this system to its best, meaning we minimize the resulting error, when we take all equations into consideration. Equation 2.1 thus turns into

$$r = Ax - b \tag{2.2}$$

The optimization is formulated as follows:

$$x^* = \arg\min_x \|Ax - b\| \tag{2.3}$$

The solution can be found by minimizing the squared residual. Minimizing means computing a solution for the first derivative set to zero.

$$\|r\|^2 = xA^TAx - 2b^TAx + b^Tb \tag{2.4}$$
$$\nabla_x \|r\|^2 = 2A^TAx - 2A^Tb = 0 \tag{2.5}$$

This further yields to the solution of the normal equation

$$A^TAx = A^Tb \tag{2.6}$$
$$x = A^T(A^TA)^{-1}b \tag{2.7}$$
$$x = A^+b \tag{2.8}$$

where equation 2.8 describes the famous Moore-Penrose pseudo inverse.

2.1.2. Quadratic Programming

Quadratic Programming handles optimization problems with linear equality or inequality constraints. Hence the name, the primal objective function to optimize is a quadratic

function. The general problem statement can be formulated as follows:

$$\frac{1}{2}x^T Qx + c^T x + r_o \tag{2.9}$$
$$\text{subject to } Dx = e \quad \in \mathcal{E} \tag{2.10}$$
$$\text{subject to } Dx \leq e \quad \in \mathcal{I} \tag{2.11}$$

Since the objective function describe a quadratic function, a least square formulation can be easily transformed into a QP. From the equation in 2.4, it can be easily seen that this equation has the same shape as equation 2.9.

We can solve general QP problems by applying Lagrange multiplier for each constraints. We formulate the Lagrangian equation by multiplying each constraint and optimize for x and λ.

$$\mathcal{L}(x, \lambda) = \frac{1}{2}x^T Qx + c^T x + r_o + \lambda^T (Dx - e) \tag{2.12}$$

The Karush-Kuhn-Tucker conditions [29] for this are given as follows

$$\nabla \mathcal{L}(x, \lambda) = 0 \tag{2.13}$$
$$d_i^T x = e_i \quad \in \mathcal{E} \tag{2.14}$$
$$d_i^T x \geq e_i \quad \in \mathcal{I} \tag{2.15}$$
$$\lambda \geq 0 \quad \in \mathcal{I} \tag{2.16}$$
$$\lambda(d_i^T x - e_i) = 0 \quad \in \mathcal{I} \tag{2.17}$$

When we look closer at the last two conditions 2.17 and 2.17, we can see that all active inequality constraints have to be satisfied. Condition 2.17 emphasizes that only constraints are valid and thus optimized, where their according λ is positive. Conditions with positive Lagrange multipliers are not considered to contribute to an optimal solution. Moreover, inequalities are called satisfied, when the optimal solution lies directly on the border. Thus, an inequality turns into an equality constraint (condition 2.17) and can be solved in a least square sense.

Considering those two conditions, we always have to find the optimal set of inequality constraints, which are active and thus can be solved in a equality like behavior. Prominent approaches are Simplex algorithms [41] and Active Set Search[42] methods. The latter will be explained in more detail in the remainder of this chapter.

2.1.3. Inverse Kinematics Control Scheme

GIK can be solved with a geometrical approach[3] just to a certain extend [45]. When the number of actuated joints increases, in particular in redundant manipulators, an numerical approach for solving the inverse kinematics problem is widely chosen [6]. The differential equation for the inverse kinematics describes a linear mapping between cartesian space and operational space

$$\dot{\boldsymbol{x}} = \boldsymbol{J}\dot{\boldsymbol{q}} \tag{2.18}$$

where $\dot{\boldsymbol{x}}$ denotes a velocity in cartesian space, whereas $\dot{\boldsymbol{q}}$ defines the robot control input vector. The Jacobian matrix $\boldsymbol{J}$ describes the linear mapping function between the two spaces. From the above formulation, we can see the similarity to the least squares equations. Thus, since a redundant manipulator describes an overdetermined system, it makes sense to solve this optimization problem with a pseudo-inverse approach.

$$\dot{\boldsymbol{q}} = \boldsymbol{J}^{+}\dot{\boldsymbol{x}} \tag{2.19}$$

Redundant manipulator implies a set of possible solutions, since the system is overdetermined and thus comprises unused DoF. Those unused DoF can be used to cascade secondary objectives without violating the first constraint [4]. In order to achieve this, we can freely multiply any arbitrary vector $\boldsymbol{q_1}$ with the according projection $\boldsymbol{P_0}$ into the nullspace of the primal objective function. Therefore, we extend equation 2.19 to

$$\dot{\boldsymbol{q}} = \boldsymbol{J}_0^{+}\dot{\boldsymbol{x}}_{\boldsymbol{0}} + \boldsymbol{P}_0\dot{\boldsymbol{q}}_1 \tag{2.20}$$

where $\boldsymbol{P}$ is defined as $\boldsymbol{I} - \boldsymbol{J}^{+}\boldsymbol{J}$.

This formulation can be included into a real time constrained control scheme. The partial velocities, solved with the above method, are integrated over time and the resulting joint positions $\boldsymbol{q}$ are set on the respective motors. The control scheme can be formulated as follows:

$$\boldsymbol{q}(t_{k+1}) = \boldsymbol{q}(t_k) + [\boldsymbol{J}(t_{k+1})^{+}\dot{\boldsymbol{x}}(t_{k+1}) + \boldsymbol{P}\dot{\boldsymbol{q}}(t_{k+1})]\Delta t \tag{2.21}$$

However, since the control scheme is mainly driven by a forward control design, it makes sense to put this control scheme in dependency of a desired cartesian position $\boldsymbol{x_d}$ and regulate the resulting error between actual solution and desired position. We formulate the error equation by incorporating the desired values

$$\dot{\boldsymbol{e}} = \dot{\boldsymbol{x}}_d - \dot{\boldsymbol{x}} = \dot{\boldsymbol{x}}_d - \boldsymbol{J}\dot{\boldsymbol{q}} \tag{2.22}$$

When we look in particular at equation 2.22, we can see a first-order differential equation. The solution as written in equation 2.19 thus becomes

$$\dot{\boldsymbol{q}} = \boldsymbol{J}^{+}\dot{\boldsymbol{e}} \tag{2.23}$$

$$= \boldsymbol{J}^{+}(\dot{\boldsymbol{x}}_d - \dot{\boldsymbol{x}}) \tag{2.24}$$

Additionally, to complement the control law, a proportional constant K is introduced as a gain parameter to optimize the convergence speed. The overall controller can be illustrated in the following diagram (figure 2.1).

2.2. Hierarchical Inverse Kinematics

Current state of the art humanoid robots possess around 30-40 DoF. This allows an execution of multiple tasks in parallel. We start our examination by firstly looking at one task.

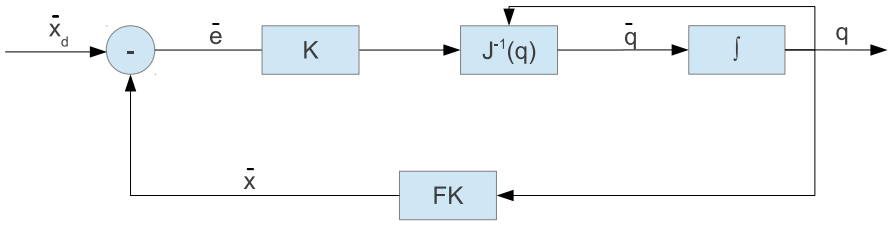


Figure 2.1.: Illustration of the Inverse Kinematics control scheme. The error is computed via the desired position x_d and the current velocity provided by the Forward Kinematics FK.

When we want to control any actuated end-effector, we consume 6 DoF in $SE(3)$ Lie group [46] for a full pose positioning. A simple manipulating task can be formulated based on a task space value $\dot{\boldsymbol{e}}$, with an unknown joint vector $\dot{\boldsymbol{q}}$ and the translating Jacobian $\boldsymbol{J} = \frac{\partial e}{\partial q}$.

$$\dot{\boldsymbol{e}} = \boldsymbol{J}\dot{\boldsymbol{q}} \tag{2.25}$$

where $\dot{\boldsymbol{q}}$ describe the solution vector in joint velocity space, which will serve as the input of the robot. The GIK formulation in equation 2.25 can be solved in a least square approach, solving the following QP to its best:

$$\dot{\boldsymbol{q}}^* = \arg\min_{\dot{q}} \|\boldsymbol{J}\dot{\boldsymbol{q}} - \dot{\boldsymbol{e}}^*\| \tag{2.26}$$

The above QP formulation is broadly used for inverse kinematics [58][5] or inverse dynamics [28]. For the remainder of this chapter, we consider a general least square approach and rename equation 2.26 into

$$x^* = \arg\min_{x} \|Ax - b\| \tag{2.27}$$

The solution for this optimization problem can be found by utilizing the pseudo-inverse approach [48], which yields to

$$x^* = A^+ b \quad \text{with } A^+ = A^T(AA^T)^{-1} \tag{2.28}$$

where A^+ denotes the Moore-Penrose pseudo-inverse [2]. Considering a set of constraints, 2.27 can be enhanced by a nullspace projection [33] to obtain a generic solution.

$$x^* = A^+ b + Px_p \tag{2.29}$$

where P denotes the projection of an arbitrary vector x_p into the nullspace of A_1. However, since we have a high number of redundant joints, we can execute multiple tasks in parallel. Thus, a task can be positioning an arm end-effector, keeping the vision system orientated towards a fixed point in 3-dimensional space or use both legs for walking. On the other hand, multiple tasks might not be feasible, since they either share common parts of the kinematic chain (e.g. torso links between the left and right arm) or interfere with each other. The idea is to implement a hierarchy of tasks, where the highest priority task will

be solved to its best in a least square approach, whereas lower prioritized tasks are solved inside the of each respective precedent nullspace.

Equation 2.29 allows us to include a secondary constraint into the QP, which does not violate the primal objective function. Considering now two tasks, we can setup a simple hierarchy based on that nullspace projection. Let A_1, b_1 be the primal objective and A_2, b_2 a secondary task, we can formulate the following hierarchical QP, substituting equation 2.29 into 2.27:

$$\begin{aligned} x_2^* = \arg\min_{x_2} \quad & \|A_2 x_2 - b_2\| \\ & \left\|A_2(A_1^+ b_1 + P_1 x_2) - b_2\right\| \\ & \left\|A_2 P_1 x_2 - (b_2 - A_2 A_1^+ b_1)\right\| \end{aligned} \tag{2.30}$$

The generic solution for this QP, can be equally as in equation 2.29 found with the pseudo-inverse:

$$x_2^* = (A_2 P_1)^+ (b_2 - A_2 A_1^+ b_1) + P_2 x_3 \tag{2.31}$$

This implies that a solution for lower stacked tasks can only be found if there exists a nullspace of a precedent task. If the above task consumes all DoF, lower placed tasks can not be considered any further.

The above equation 2.31 is recursively formulated in [50] for n hierarchically stacked tasks.

$$x_n^* = \sum_{k=1}^{n} (A_k P_{k-1})^+ (b_k - A_k A_{k-1}^+ b_{k-1}) + P_n x_{n+1} \tag{2.32}$$

2.3. Inequality Constraints

The above presented hierarchy describes a equality constrained Quadratic Program (eQP). This enables a simultaneous execution of multiple tasks with equality constraints, such as positioning tasks. However, in order to implement safety mechanism, no hard constraints but solution spaces are necessary, which immediately demand for inequality constraints. Safety mechanisms can be such as an active elbow-field, keeping the elbow above a certain threshold, balancing the center of mass (CoM) inside a polygon or avoiding collision with obstacles, as it is the topic of this work.

We can formulate an inequality Quadratic Program (iQP) in a classical manner:

$$x^* = \arg\min_{x} \|Ax - b\| \tag{2.33}$$

$$\text{subject to } Cx < d \tag{2.34}$$

These kind of iQP are often formulated by introducing a slack variable w, which indicates the level of violation [24].

$$x^*, w^* = \arg\min_{x,w} \|w\| \tag{2.35}$$

$$\text{subject to } Cx < d + w \tag{2.36}$$

Let us assume that $Cx < d$ fulfills all the KKT [29] conditions. We can then denote this an optimal solution set and easily solve this iQP as if it were an eQP. This means, that in an optimal set of inequalities, the solution satisfies all active constraints [11].

The main idea of solving such an iQP is finding the set of constraints, which form the optimal set. Those algorithms are mainly referred as Active Set Search [42] and often used to solve inequality constraint QPs. The algorithms starts with an initial guess $\boldsymbol{x}_0$ and working set $\boldsymbol{W}$ filled with activated constraints. The first step is to solve the iQP like an eQP and find the KKT optimal conditions $\boldsymbol{x}_{t=1}, \boldsymbol{\lambda}_{t=1}$. When one or many Lagrange multiplier are negative, the currently active set is not optimal, which means there are unnecessary constraints. The negative constraint λ_k is dismissed and the algorithm starts over again with $\boldsymbol{x}_0$ and $\{\boldsymbol{W} \setminus \lambda_k\}$ in order to solve $\boldsymbol{x}_{t=2}, \boldsymbol{\lambda}_{t=2}$.

Once a solution is found, where $\boldsymbol{\lambda}$ is strictly positive, we perform a search step. This is by best practice performed in the direction of $\boldsymbol{v} = \boldsymbol{x}_1 - \boldsymbol{x}_2$.

$$\boldsymbol{x}_\mathbf{1}^* = \boldsymbol{x}_\mathbf{1} + p\boldsymbol{v} \quad \text{with } p \in [0; 1]$$

We have to prove the new step for feasibility. We again solve the QP and activate all constraints, which are violated on the update step. The algorithm converges when all constraints are feasible and contain non-negative Lagrange multiplier.

We can see that the recursive formulation in equation 2.32 is not applicable towards inequality constraints. The problem is that we can not easily find a nullspace projector P, as this can be a n-dimensional semi-infinite space. [23] presents a hierarchical preserving way to solve inequality constraints in the form of equation 2.35. Let us consider a primal objective with two inequality constraints, which are aligned in hierarchical manner.

$$\arg\min_{x_2,w_2} \|w_2\| \tag{2.37}$$

$$\text{subject to } A_1 x < b_1 + w_1^* \tag{2.38}$$

$$\text{subject to } A_2 x < b_2 + w_2 \tag{2.39}$$

The above formulation shows the solution approach of recursively concatenated iQP. The first inequality in 2.37 provides a valid solution x with respect to the least square solution of the primal objective function. With this, it is secured that the solution of x_1 does not change the solution in x. This behavior can be concatenated recursively.

Recomputing the eQP constantly for finding the optimal active set is computational expensive [11]. It is mainly expensive, because activating and deactivating constraints forces the recomputation of the complete stack. Furthermore, each constraint is solved on its according level and all levels beneath it. When the stack contains n levels, then in particular the highest level is solved n times. This is not feasible in terms of whole body motion control, as it would probably exceed all real time update cycles on a humanoid robot. The HCOD introduces an efficient matrix decomposition based on an existing Complete Orthogonal Decomposition (COD). The improvements exploits a methods, which depends only on the amount of DoF and not on the constraints. In essence, the HCOD uses a intelligent way to improve the computation speed of the Active Set Search. Furthermore, HCOD implies ways to solve the complete stack in one cycle, without recursively backtrack all beneath placed stacks. The results of HCOD exceed existing solutions by an order of magnitude in computation time. This allows a powerful setup of tasks to be executed in parallel on low powered computers. The mathematical insights of this are verbosely given in [12] and exceed the scope of this thesis.

CHAPTER 3

COLLISION AVOIDANCE

In the following chapter, we introduce our proposed method for collision avoidance. We implemented an inequality task inside the SoT, which is placed as a high priority task. This task handles all collisions, but still restricts the execution of lower priority tasks as little as possible. The collision avoidance task, introduced in the following, is based on three major concepts.

1. Closest point pair calculation between two collision objects
2. Damping the velocity along the normal vector, spanning from the closest point pair
3. Calculating the Jacobian for every closest point

3.1. Closest Point Calculation

Closest point pairs are used in various algorithms for collision avoidance [9][22]. Hereby, a Gilbert-Johnson-Keerthi (GJK) approach is frequently used to compute the proximity and closest points for collision detection [57]. It is easy to understand, that preventing the closest points of two colliding objects is sufficient enough for avoiding the complete object to collide. However, this is constrained with the condition, that there is always one unique pair of closest points [15].

3.1.1. Collision Geometry - Mesh

Most common robots come along with a description of their collision geometry. In order to make the collision geometry as accurate as possible, polygon meshes are widely used to represent the geometry of the robot. The interested reader is referred to [56] for more information.

Although the calculation between two meshes might fulfill the requirement of having a unique point pair between two collision objects, severe discontinuities occur inside a two dimensional space. Depending on the alignment of the vertices inside the mesh, the closest points can jump in X as well as Y direction, when two meshes appear to be coplanar to each other. In this case, there is ambiguity in the placement of the two points. Figure 3.1 illustrates this behavior. The discontinuities shown here are linearly propagated to the velocity of the moving body part, which finally results in a heavy jump of velocity.
The discontinuities occur quite frequently in two aspects. The points can vary inside one pair of vertices as illustrated in figure 3.1. Additionally, depending on the shape of the mesh, multiple vertices can be projected towards the same pose, respectively a high number of normal vectors of each vertices (see figure 3.2). This makes a closest point calculation for meshes infeasible.

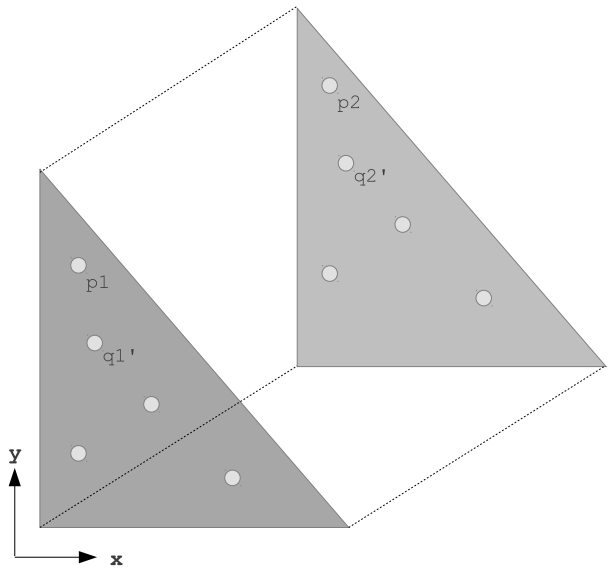


Figure 3.1.: Discontinuity of closest point calculation for two vertices inside meshes. Parallelism of two vertices produces an ambiguity of the point pair. As depicted here, multiple solutions exists such as $p1$, $p2$ as well as $p1'$, $q2'$.

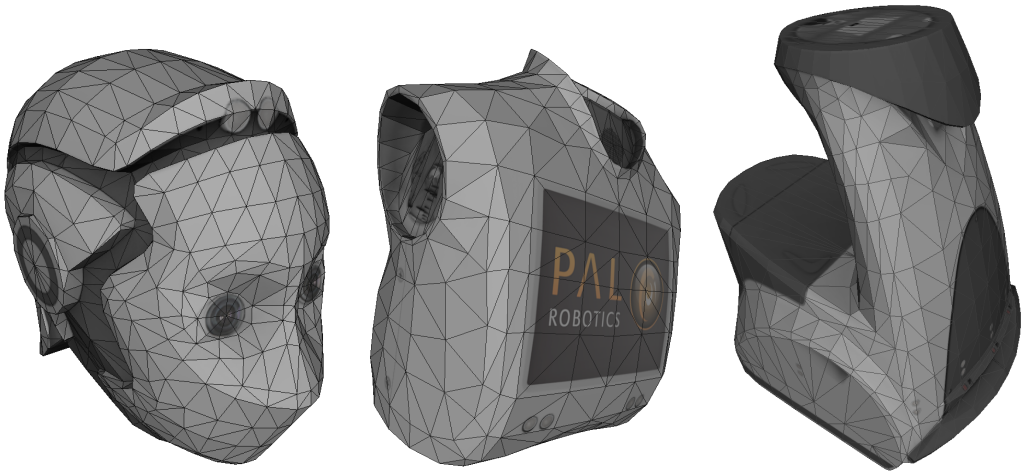


Figure 3.2.: Example for high resolution meshes. It can be easily seen, that a high number of the same normal vectors exist on the meshes. This makes a parallel configuration quite likely and the closest points can jump discontinuously.

3.1.2. Collision Geometry - Capsule

To overcome the discontinuities mentioned in the previous section, we implemented a complete capsule decomposition for this work. A capsule describes a 3-dimensional geometry object, which consists of a cylinder along a 3-dimensional main axis and a length l and radius r. In difference to the cylinder, each capsule comprises a half sphere at both ends of the cylinder. Since capsules are pseudo convex collision geometries, the probability of discontinuities is relatively low. There exists exactly one configuration, where ambiguity can occur. Essentially this happens, when the main axis of both capsules are parallel. Figure 3.3 expresses two possible configuration for closest point pairs. It can be seen that as long as the main axis are not parallel aligned, both capsules can be treated as strictly convex objects. This convexity results in a unique point pair.
Furthermore, a capsule decomposition allows a smoother continuous trajectory compared to mesh to mesh calculation. In the case of parallel main axis, the dimension of possible discontinuities compared to meshes is limited to one dimension along the main-axis

$$\boldsymbol{p_i} + l_i \boldsymbol{n} \quad i \in [CP_1, CP_2], l \in \mathbb{R}, \boldsymbol{n} \in \mathbb{R}^3 \tag{3.1}$$

where p_i describes the start point of the cylinder with $z = 0$, l_i the length of the capsule and n the orientation of the main axis. Note that in this specific case, we do not differentiate n_{C1} and n_{C2} as both capsules are considered to be parallel and the z component is pointing along the axis.

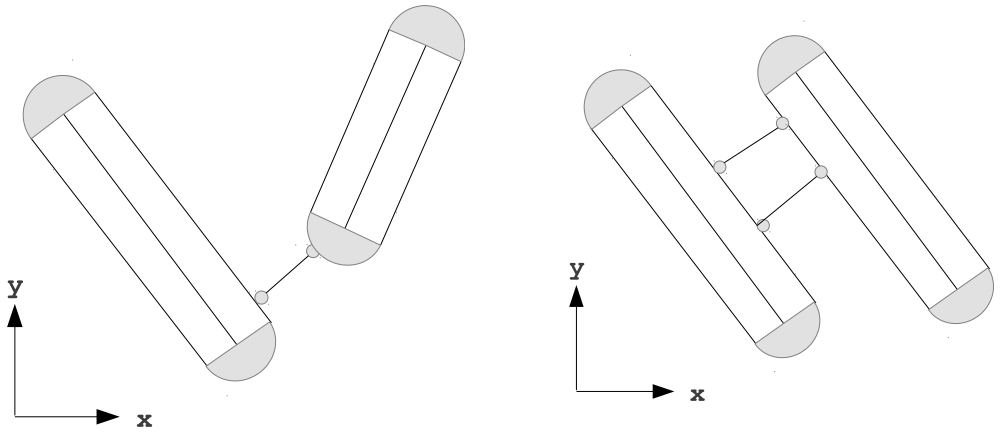
Given this definition, the distance of possible discontinuities can be formulated as

$$\boldsymbol{p_1} + l_1\boldsymbol{n} = \boldsymbol{p_2} + l_2\boldsymbol{n} \tag{3.2}$$

$$\boldsymbol{p_1} + l_1\boldsymbol{n} - (\boldsymbol{p_2} + l_2\boldsymbol{n}) = 0 \tag{3.3}$$

$$(p_{z1} + l_1) - (p_{z2} + l_2) \tag{3.4}$$

where in equation 3.4, only the Z component has to be considered. X and Y are equal, since the two capsules are parallel.



(a) Unique solution (b) Parallel configuration

Figure 3.3.: a) shows a unique solution of a closest point pair. b) shows two capsules in a parallel configuration. Note that in comparison to meshes, the ambiguity spans only along the main axis of the capsules. In a continuous motion of both capsules, these discontinuities are from no further importance, because the probability is rather low.

3.1.3. Closest Point Pair for Capsule-Capsule

We concluded in the previous section 3.1.2, that a capsule decomposition of collision objects is to be preferred over meshes, because of the reduction of discontinuities. In this section, we describe the algorithm for the actual computation of the closest point pair. The presented algorithm is a derivative of the closest points based on two line segments as it is verbosely described in [10]. Since we define a capsule to be a finite line segment in $\mathbb{R}^3$ with spatial radius information, we can modify the line segment algorithm to fit the calculated point on the surface of the capsule.

Infinite Lines

We start our calculation by looking at the closest point calculation of two lines. According to [10], we formulate a line pair in $\mathbb{R}^3$ as follows.

$$\begin{aligned} LS_1(s) &= \boldsymbol{p_1} + s\boldsymbol{d_1} \\ LS_2(t) &= \boldsymbol{p_2} + t\boldsymbol{d_2} \\ \textit{where} &\quad \boldsymbol{d_i} = \boldsymbol{q_i} - \boldsymbol{p_i} \end{aligned} \tag{3.5}$$

The start- and endpoint are declared as $\boldsymbol{p}$ and $\boldsymbol{q}$ in $\mathbb{R}^3$, respectively. $\boldsymbol{d}$ indicates the directional vector between $\boldsymbol{p}$ and $\boldsymbol{q}$. We find a solution for a closest point pair s, t, whose vector

$$\boldsymbol{v}(s,t) = LS1 - LS2 \tag{3.6}$$

gets minimal in length. Example is given in figure 3.4(a).

The shortest vector $\boldsymbol{v}(s,t)$ can be examined as the scalar product of the two lines. Since the vector has to be perpendicular to both lines, the following equation system for the scalar product can be formulated:

$$\begin{aligned} \boldsymbol{d_1}\boldsymbol{v}(s,t) &= 0 \\ \boldsymbol{d_2}\boldsymbol{v}(s,t) &= 0 \end{aligned} \tag{3.7}$$

we solve this equation system by substituting $\boldsymbol{v}(s,t)$ with the definition of 3.6. Further steps are invoked:

$$\begin{aligned} \boldsymbol{d_1}(LS_1 - LS_2) &= \boldsymbol{d_1}((\boldsymbol{p_1} - \boldsymbol{p_2} + s\boldsymbol{d_1} - t\boldsymbol{d_2}) = 0 \\ \boldsymbol{d_2}(LS_1 - LS_2) &= \boldsymbol{d_2}((\boldsymbol{p_1} - \boldsymbol{p_2} + s\boldsymbol{d_1} - t\boldsymbol{d_2}) = 0 \end{aligned} \tag{3.8}$$

$$\begin{aligned} (\boldsymbol{d_1}\boldsymbol{d_1})s - (\boldsymbol{d_1}\boldsymbol{d_2})t &= -\boldsymbol{d_1}k \\ (\boldsymbol{d_1}\boldsymbol{d_2})s - (\boldsymbol{d_2}\boldsymbol{d_2})t &= -\boldsymbol{d_2}k \end{aligned} \tag{3.9}$$

where $k = \boldsymbol{p_1} - \boldsymbol{p_2}$. Using Cramer's rule [7] for solving 2x2 linear equation system finally yields the solution for s and t:

$$\begin{aligned} s &= (bf - ce)/d \\ t &= (af - be)/d \end{aligned} \tag{3.10}$$

where $a = \boldsymbol{d_1^2}, b = \boldsymbol{d_1}\boldsymbol{d_2}, c = \boldsymbol{d_2^2}, e = \boldsymbol{d_1}k, f = \boldsymbol{d_2}k, d = \boldsymbol{d_1^2}\boldsymbol{d_2^2} - (\boldsymbol{d_1}\boldsymbol{d_2})^2$
Special care has to be taken in case $d = 0$, which means that both lines are parallel.

Line Segments

Compared to lines, line segments are of finite length. This necessitates a case differentiation of the spatial pose relation between two line segments. For this, we simply restrict the

length along the directional vector in the range from 0 to 1. Thus, we modify equation 3.5 according to:

$$LS_i(s) = \boldsymbol{p_i} + s\boldsymbol{d_i} \quad \textit{where } \boldsymbol{d_i} = \boldsymbol{q_i} - \boldsymbol{p_i}, 0 \leq s \leq 1 \tag{3.11}$$

With this formulation, we cannot apply the solution used in the previous section, as there might not be a perpendicular vector v for all configurations. This situation happens, when at least one line segments lies outside the line segment of the other. In the following, we examine four different cases as illustrated in figure 3.4.

When a configuration occurs as shown in figure 3.4(a), we can apply the solutions as if the two line segments were infinite lines. When one closest point CP_1 lies outside its line segments, yet inside the segment of the corresponding other line, we can clamp CP_1 to the end of the line segment and compute the minimum distance to LS_2, in order to find CP_2. This behavior is illustrated in figure 3.4(b). In case both closest points lie outside the corresponding line segments (see figure 3.4(c), the procedure of 3.4(b) has to be repeated twice. We first clamp CP_1 to the end point and calculate $\hat{CP_2}$. As depicted in figure 3.4(c), this point lies outside LS_2 and results in a invalid point. To overcome this issue, we clamp $\hat{CP_2}$ to the endpoint of LS_2 and repeat the computation now inverse. We recompute CP_1 and see that this lies now inside LS_1. Figure 3.4(d) illustrates the very extreme, where both points are clamped to their respective end points.

Finally, the solving process is separated in two parts. Firstly, we have to decide for the clamping of the respective points. This case study is examined according to the above description. Secondly, once the points are clamped, we can mathematically solve for the closest points.
We have a point LS_2 lying on the second segment:

$$LS_2(t) = \boldsymbol{P_2} + t\boldsymbol{d_2} \tag{3.12}$$

The first point LS_1 is found via:

$$\begin{aligned} LS_1(s) &= \boldsymbol{P_1} + s\boldsymbol{d_1} \\ s &= (LS_2(t) - \boldsymbol{P_1})\boldsymbol{d_1}/\boldsymbol{d_1}\boldsymbol{d_1} \\ &= (\boldsymbol{P_2} + t\boldsymbol{d_2} - \boldsymbol{p_1}\boldsymbol{d_1}/\boldsymbol{d_1}\boldsymbol{d_1} \end{aligned} \tag{3.13}$$

Obviously, this formulation also determines LS_2, given LS_1.

Capsules

In the beginning of this chapter, we defined a capsule as a line segment with a spatial radius. For the closest point computation on two capsules, we enhance the algorithm for line segments slightly to project the closest on the border of the capsule.
As we know the two closest points $\hat{\boldsymbol{CP_1}}$ and $\hat{\boldsymbol{CP_2}}$ of the two respective line segments, we can use this information to calculate the directional vector between them. The projection

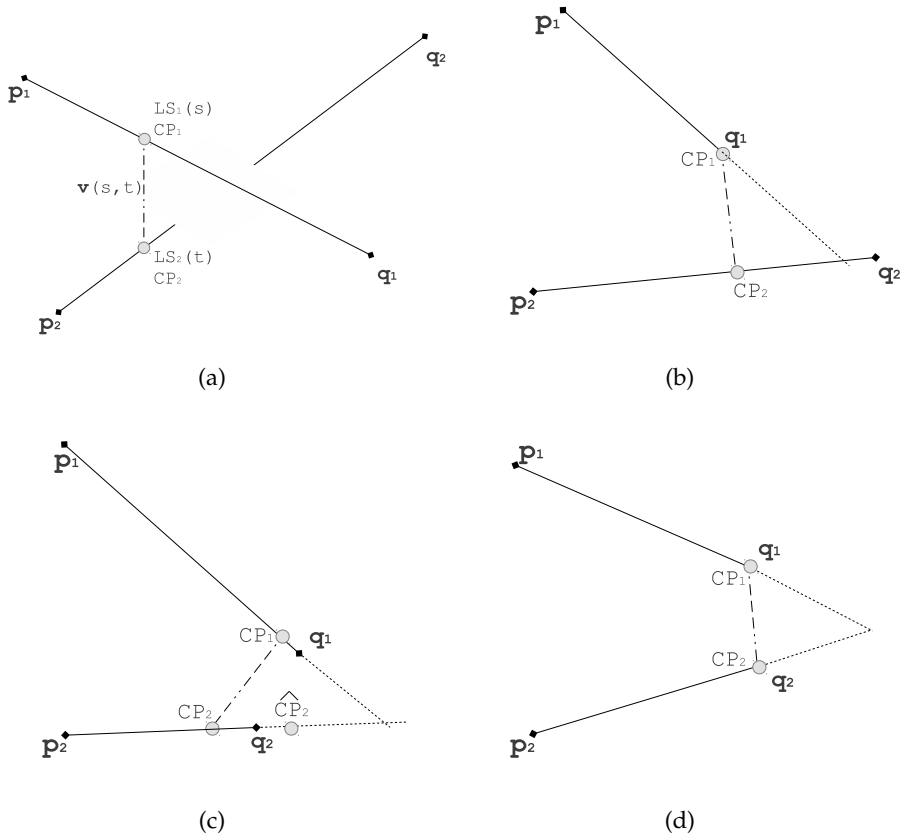


Figure 3.4.: Relative spatial poses between two line segments. a) closest point pair lies inside both segments. b) endpoint q_1 is a closest point, the second closest point lies on the line segment. c) both closest points would lie outside their corresponding line segments. Here, a recursive clamping has to be implemented. d) both ending points p_1 and p_2 are closest points.

is to the border of the capsule is done by shifting the closest points along this vector by the radius. Figure 3.5 shows this projection.

We can formulate this solution, given the two closest points $\hat{\boldsymbol{CP_1}}, \hat{\boldsymbol{CP_2}}$. We calculate the vector $\boldsymbol{v}$:

$$\boldsymbol{v} = \hat{\boldsymbol{CP_2}} - \hat{\boldsymbol{CP_1}} \tag{3.14}$$

The shifting is done via:

$$\begin{aligned} \boldsymbol{CP_1} &= \hat{\boldsymbol{CP_2}} + r_1\boldsymbol{v} \\ \boldsymbol{CP_1} &= \hat{\boldsymbol{CP_1}} - r_2\boldsymbol{v} \end{aligned} \tag{3.15}$$

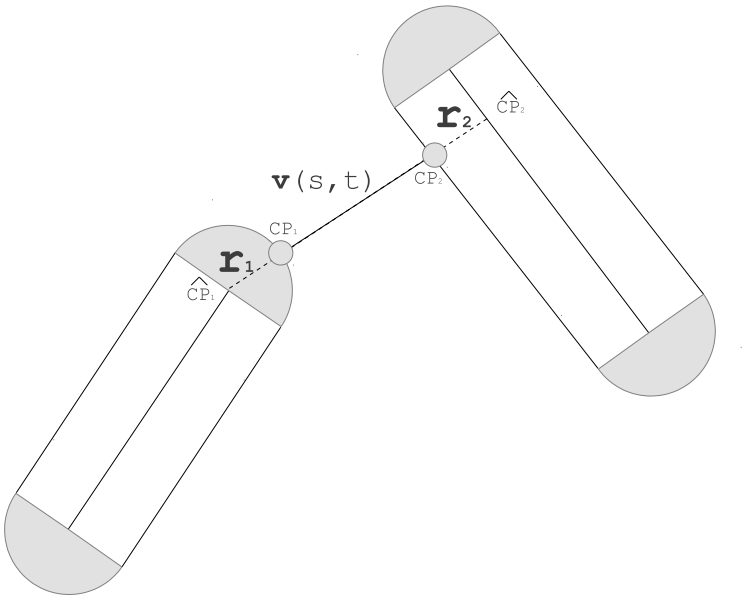


Figure 3.5.: Projection of the two closest points $\hat{\boldsymbol{CP_1}}$ and $\hat{\boldsymbol{CP_2}}$ to the border of the capsule. $\boldsymbol{v(s,t)}$ describes the directional vector along which the projection is done with length r_i.

3.2. Velocity Damping

In section 2 we pointed out that every task has to be designed in the form of

$$\boldsymbol{Ax - b} \quad \boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^m \tag{3.16}$$

In this section, we introduce the task formulation for collision avoidance. The collision avoidance task restricts any possible movement along the unitvector between the closest point pair, we calculated in the previous section. The presented approach was firstly introduced in [17] and reformulated inside the SoT [22]. As we have seen in equation 2.26, controlling a body part in cartesian space corresponds to the IK formulation

$$\boldsymbol{J(p,q)\dot{q} - \dot{x}} \quad \boldsymbol{J} \in \mathbb{R}^{m \times n}, \boldsymbol{\dot{q}}, \boldsymbol{\dot{x}} \in \mathbb{R}^n \tag{3.17}$$

where $\boldsymbol{J(p,q)}$ names the Jacobian of point $\boldsymbol{p}$ at joint vector position $\boldsymbol{q}$, the resulting joint velocities $\boldsymbol{\dot{x}}$ and the cartesian goal velocity. This formulation solves for all available DoF inside the hierarchy. In order to restrict the solution for one direction, we modify the above formulation in two aspects.

- The Jacobian is projected onto the unitvector spanned by the closest point pair
- The task is formulated as an inequality, which introduces a lower boundary for the proximity (distance) of the two body parts

3.2.1. Jacobian Projection

The configuration setup according to [17] is depicted in figure 3.6. The figure shows a close up view of a capsule pair, with closest points $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$. Hereby, we consider the point $\boldsymbol{p_1}$ as the moving body part, whereas $\boldsymbol{p_2}$ is meant to be fixed[1] in $\mathbb{R}^3$ as a collision object. The unitvector $\boldsymbol{n}$ is defined as

$$\boldsymbol{n} = \frac{\boldsymbol{p_1} - \boldsymbol{p_2}}{\|\boldsymbol{p_1} - \boldsymbol{p_2}\|} \tag{3.18}$$

The distance is computed according to the euclidean L2-Norm:

$$d = \|\boldsymbol{p_1} - \boldsymbol{p_2}\| = \sqrt[2]{(\boldsymbol{p_1} - \boldsymbol{p_2})^2} \tag{3.19}$$

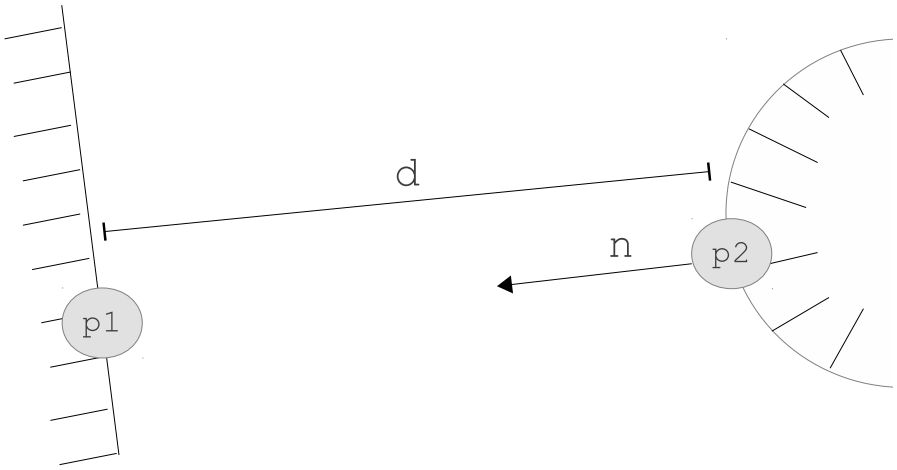


Figure 3.6.: A close up view of two capsules. The unitvector n is pointing from p_2 to p_1. The distance d corresponds to the euclidean L2-norm of p_1 and p_2.

To begin with, we modify the task definition in equation 3.17 to restrict only the projection along n. Thus, the new task formulation becomes

$$\boldsymbol{n^T J(p_1, q)\dot{q} - \dot{x}} \quad , \boldsymbol{n} \in \mathbb{R}^m, \boldsymbol{J} \in \mathbb{R}^{m \times n}, \boldsymbol{\dot{q}}, \boldsymbol{\dot{x}} \in \mathbb{R}^m \tag{3.20}$$

where $\boldsymbol{n^T}$ describes the scalar product of $\boldsymbol{n}$ and $\boldsymbol{J}$. Informally spoken, we can treat the scalar product as the projection of the Jacobian onto the unitvector pointing towards the collision center. Note that we annotate the Jacobian parameters with respect to p_1. With this, we indicate, that the Jacobian has to be calculated in respect to the closest point p_1. The next section 3.3 gives a more detailed examination on it.

[1] We establish the task with this convention of $\boldsymbol{p_2}$ being fixed in space. However, the implementation necessarily allows both points to be moving.

3.2.2. Inequality Formulation

To avoid a possible collision between two capsules, we have to ensure that the distance between them will always be larger than zero. To guarantee a minimal distance threshold, we transform the task in equation 3.20 into an inequality constraint. Furthermore, we introduce a lower bound with respect to the actual distance. The outer circle in figure 3.7 represents this lower bound, where the minimal proximity of the point pair $\boldsymbol{p_1}$,$\boldsymbol{p_2}$ has to be larger than d_s. The solution space for this inequality task is formulated in respect to the

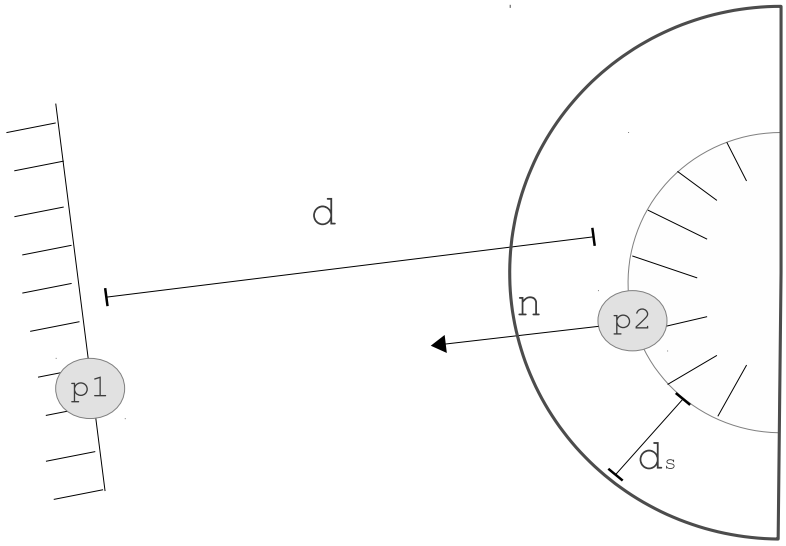


Figure 3.7.: To ensure a minimal proximity between two capsules, a security distance d_s is introduced. The closest point $\boldsymbol{p_1}$ has to be outside the distance zone.

current distance d. Thus, the equation 3.20 evolves to an inequality constraint

$$\boldsymbol{n}^T \boldsymbol{J}(\boldsymbol{p_1}, \boldsymbol{q})\dot{\boldsymbol{q}} \geq -\epsilon \frac{d - ds}{\Delta t} \quad , \epsilon, d, d_s, \Delta t \in \mathbb{R} \tag{3.21}$$

where $d - d_s$ calculates the distance along $\boldsymbol{n}$. The velocity towards the collision center undergoes a damped course, hence the name velocity damping. We stress out the Δt in equation 3.21 to demonstrate the calculation in the velocity domain. In the implementation although, this value is not available as a parameter, since the control gain value ϵ has to be tuned for each application, which may compensate for Δt.
We can see that the velocity converges to zero with decreasing proximity between the closest point pair.

$$\lim_{d \to d_s} -\epsilon \frac{d - ds}{\Delta t} = 0 \tag{3.22}$$

We can now formulate the overall optimization problem as a task space formulation where a position $\boldsymbol{x_d}$ is specified as a goal position and $\boldsymbol{x}$ the current cartesian position. The optimization of this task is now subject to the velocity damping constraint developed in this section. The verbose formulation of the QP looks as the following:

$$\min_{\dot{\boldsymbol{q}}} \rightarrow \|\boldsymbol{J}(\boldsymbol{p_x}, \boldsymbol{q})\dot{\boldsymbol{q}} - \dot{\boldsymbol{x}}\|^2$$

$$\min_{\dot{\boldsymbol{q}}} \rightarrow \|\boldsymbol{J}(\boldsymbol{p_x}, \boldsymbol{q})\dot{\boldsymbol{q}} - \frac{(\boldsymbol{x} - \boldsymbol{x_d})}{\Delta t}\|^2 \tag{3.23}$$

$$\text{subject to} \quad \boldsymbol{n}^T\boldsymbol{J}(\boldsymbol{p_1}, \boldsymbol{q})\dot{\boldsymbol{q}} \geq -\epsilon\frac{d - ds}{\Delta t} \tag{3.24}$$

One note to the above formulation: It might look counter-intuitive at a first glance to design the task like this. One would presume a twist in the directions of $\boldsymbol{n}$ as well as the preceding minus sign in equation 3.21. It is more intuitive to assume $\boldsymbol{n}$ to point from $\boldsymbol{p_1}$ to $\boldsymbol{p_2}$. This modulation comes directly from the architecture of the hierarchical solver as a first-order control system. We review again the derivation of a task in cartesian space, which is depicted in the diagram 2.1.

$$\begin{aligned}
\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} &= \dot{\boldsymbol{x}} \\
\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} &= K\frac{\boldsymbol{x} - \boldsymbol{x_d}}{\Delta t} && (3.25)\\
\boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} &= -K\frac{\boldsymbol{x_d} - \boldsymbol{x}}{\Delta t} && (3.26)\\
\dot{\boldsymbol{x}} &\equiv -\hat{K}\boldsymbol{x_e} \quad \text{, where } \hat{K} = \frac{k}{\Delta t} && (3.27)
\end{aligned}$$

We can see from the above formulation that between equation 3.25 and equation 3.26 the sign changed as the formulation of the error between $\boldsymbol{x}$ and $\boldsymbol{x_d}$ swapped. Finally, 3.27 results in a common first-order control system formulation. The velocity damping task obeys this formulation.

3.2.3. Integration into Hierarchy Solver

The QP formulation represents an example for one collision pair. Equation 3.24 describes an inequality constraint for one closest point pair $\boldsymbol{p_1}$,$\boldsymbol{p_2}$ and the according distance d.
Obviously regarding the self-collision avoidance, we have to extend this to integrate multiple velocity damping constraints. Depending on the application, it might be important on which level these inequality constraints are placed inside the solver hierarchy. This further depends which relative level the damping constraints have to each other inside the hierarchy. We can consider two cases:

- Constraints are on the same level
- Constraints are on different level

Same Level

Looking at self-collision avoidance, it is necessary that all collision pairs are on an equal level inside the solver hierarchy. This comes with no surprise, as one would expect the left arm being equally treated as the right arm in terms of avoiding self-collision.
To achieve this, we can concatenate the formulation given in 3.24. For multiple collision points inside one task, the formulation evolves to:

$$\boldsymbol{n_i}^T \boldsymbol{J}(\boldsymbol{p_{1,i}}, \boldsymbol{q})\dot{\boldsymbol{q}} \geq -\epsilon_i \frac{d_i - d_{s,i}}{\Delta t} \quad \forall i \in \{1, \ldots, k\} \tag{3.28}$$

where i iterates over all k available collision pairs, which are considered to be on the same level. Written in matrix notation, this yields to

$$\begin{aligned} \begin{pmatrix} \boldsymbol{n_1}^T \boldsymbol{J_1} \\ \vdots \\ \boldsymbol{n_k}^T \boldsymbol{J_k} \end{pmatrix} \dot{\boldsymbol{q}} &\geq \begin{pmatrix} \dot{\boldsymbol{d}}_{\boldsymbol{1}} \\ \vdots \\ \dot{\boldsymbol{d}}_{\boldsymbol{k}} \end{pmatrix} \\ \boldsymbol{N}\dot{\boldsymbol{q}} &\geq \boldsymbol{D} \\ \textit{where } \boldsymbol{J_i} &= \boldsymbol{J}(\boldsymbol{p_{1,i}}, \boldsymbol{q}) \\ \textit{and } \dot{\boldsymbol{d}}_{\boldsymbol{i}} &= \frac{d_i - d_{s,i}}{\Delta t} \end{aligned} \tag{3.29}$$

Again the complete QP respecting all constraints at the same level yields:

$$\begin{aligned} \min_{\dot{\boldsymbol{q}}} \rightarrow \|\boldsymbol{J}(\boldsymbol{p_x}, \boldsymbol{q})\dot{\boldsymbol{q}} - \dot{\boldsymbol{x}}\|^2 \\ \text{subject to} \quad \boldsymbol{N}\dot{\boldsymbol{q}} \geq \boldsymbol{D} \end{aligned} \tag{3.30}$$

Different Level

There may be applications, where multiple hierarchy steps are feasible. One could think of setting priority to self-collision avoidance over external collision avoidance. In this case, no further modifications have to be implemented for the task itself. This approach can simply be realized by stacking the two instances of the velocity damping task on top of each other.

3.3. Jacobian Calculation for Closest Point

The task formulation examined in the previous section 3.2 constrains one closest point pair. Hence, we can see in equation 3.21 the subscript for $\boldsymbol{p}$ as a parameter for the Jacobian $\boldsymbol{J}$.

We concluded in section 3.1 that based on the pseudo convexity of capsules as collision geometries, the closest point pairs move along a differentiable trajectory. On the same hand, this means that for each iteration of the solver, a new Jacobian has to be computed since we parameterize the Jacobian with $\boldsymbol{p}, \boldsymbol{q}$

$$\boldsymbol{J} \equiv \boldsymbol{J}(\boldsymbol{p}, \boldsymbol{q}) \equiv \boldsymbol{J}(\boldsymbol{p}(t), \boldsymbol{q}(t)) \tag{3.31}$$

3.3.1. Jacobian Calculation for Rigid Body Parts

To begin with, we introduce a short review for the general case of computing the Jacobian matrix for rigid body parts. The experienced reader might freely skip this section. During this section, we mainly stick to the notation convention as suggested in [45].

Forward kinematics (FK) describes the function f of translating a n-dimensional joint position vector q into 6-dimensional cartesian poses. Similarly, *Differential* kinematics maps joint velocities to cartesian velocities, namely linear velocity $\dot{\boldsymbol{p}}$ and angular velocity $\boldsymbol{\omega}$. Hereby, there exists a linear relation between joint velocities and cartesian velocities:

$$\begin{aligned} \boldsymbol{P} &= f(\boldsymbol{q})\boldsymbol{q} \\ \dot{\boldsymbol{P}} &= \frac{\delta f(\boldsymbol{q})}{\delta \boldsymbol{q}}\dot{\boldsymbol{q}} \\ \dot{\boldsymbol{P}} = \begin{bmatrix} \dot{\boldsymbol{p}} \\ \boldsymbol{\omega} \end{bmatrix} &= \boldsymbol{J}\dot{\boldsymbol{q}} \\ \textit{where} \qquad & \boldsymbol{P} \in \mathbb{R}^{6\times n}, \boldsymbol{J} \in \mathbb{R}^{6\times n}, \dot{\boldsymbol{q}} \in \mathbb{R}^{n} \end{aligned} \tag{3.32}$$

From the FK, we know the following relation:

$$\boldsymbol{p_i} = \boldsymbol{p_{i-1}} + \boldsymbol{R}_{i-1}\boldsymbol{r}^{\boldsymbol{i-1}}_{\boldsymbol{i-1,i}} \tag{3.33}$$

where $\boldsymbol{p_i}, \boldsymbol{p_{i-1}}$ denote the current position vector of frame origin i and $i-1$, respectively. $\boldsymbol{R}_{i-1}$ describes the rotation matrix of frame $i-1$, whereas $\boldsymbol{r}^{\boldsymbol{i-1}}_{\boldsymbol{i-1,i}}$ denotes the relative translation from frame origin $i-1$ to i. The relation is illustrated in 3.8

Linear Velocity

For obtaining the according linear velocity formulation, we can take the first derivative of the above equation 3.33. This yields into:

$$\begin{aligned} \dot{\boldsymbol{p}}_{\boldsymbol{i}} &= \dot{\boldsymbol{p}}_{\boldsymbol{i-1}} + \boldsymbol{R}_{i-1}\dot{\boldsymbol{r}}^{\boldsymbol{i-1}}_{\boldsymbol{i-1,i}} + \boldsymbol{\omega}_{i-1} \times \boldsymbol{R}_{i-1}\boldsymbol{r}^{\boldsymbol{i-1}}_{\boldsymbol{i-1,i}} \\ &= \dot{\boldsymbol{p}}_{i-1} + \boldsymbol{v}_{i-1,i} + \boldsymbol{\omega}_{i-1} \times \boldsymbol{r}_{i-1,i} \end{aligned} \tag{3.34}$$

Hereby, the linear velocity is split into translational velocity $\boldsymbol{v}$ and rotational linear velocity $\boldsymbol{\omega}_{i-1} \times \boldsymbol{r}_{i-1,i}$ between two adjacent links.

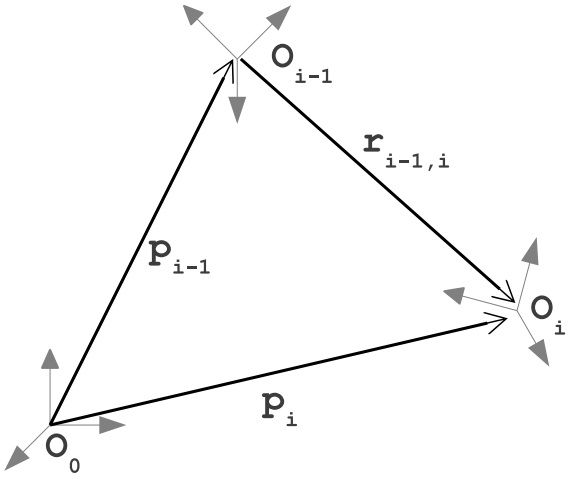


Figure 3.8.: Relative position between two origin frames expressed in a common base frame.

Angular Velocity

The angular velocity can be obtained from the rotation matrix composition

$$\boldsymbol{R_i} = \boldsymbol{R}_{i-1}\boldsymbol{R}_i^{i-1} \tag{3.35}$$

It can be shown in [45] and [49], by useful exploiting the skew matrix property, taking the first derivative of equation 3.35 yields to the angular velocity formulation:

$$\begin{aligned} \boldsymbol{\omega}_i &= \boldsymbol{\omega}_{i-1} + \boldsymbol{R}_{i-1}\boldsymbol{\omega}_{i-1,i}^{i-1} \\ &= \boldsymbol{\omega}_{i-1} + \boldsymbol{\omega}_{i-1,i} \end{aligned} \tag{3.36}$$

Furthermore, the relations examined in equation 3.33 and 3.35 differ depending on the type of joint. We have to differentiate the velocity decompositions depending whether we are considering a revolute or prismatic joint:

Revolute Joint

According to the Denavit-Hartenberg Convention (DHC), the axis of rotation is z [21]. Hence, the angular velocity simplifies to

$$\boldsymbol{\omega}_{i-1,i} = \dot{\theta}_i \boldsymbol{z}_{i-1} \tag{3.37}$$

where $\dot{\boldsymbol{\theta}}$ denotes the rotational joint speed. Using this, we calculate further the linear velocity:

$$\boldsymbol{v}_{i-1,i} = \boldsymbol{\omega}_{i-1,i} \times \boldsymbol{r}_{i-1,i} \tag{3.38}$$

Once we computed $\boldsymbol{\omega}_{i-1,i}$ and $\boldsymbol{v}_{i-1,i}$, we finally obtain:

$$\boldsymbol{\omega} = \boldsymbol{\omega}_{i-1} + \dot{\theta}_i \boldsymbol{z}_{i-1} \tag{3.39}$$
$$\dot{\boldsymbol{p}}_i = \dot{\boldsymbol{p}}_{i-1} + \boldsymbol{\omega}_i \times \boldsymbol{r}_{i-1,i} \tag{3.40}$$

Prismatic Joint

Equally to the revolute joints, we have to reconsider the equations of velocity for prismatic joints. Since the prismatic joint has no rotational part, the angular velocity equals zero

$$\boldsymbol{\omega}_{i-1,i} = \mathbf{0} \tag{3.41}$$

Also for prismatic joint, the axis of movement according to DHC is along z. The linear velocity results to

$$\boldsymbol{v}_{i-1,i} = \dot{d}_i \boldsymbol{z}_{i-1} \tag{3.42}$$

The final velocities are computed respectively:

$$\boldsymbol{\omega} = \boldsymbol{\omega}_{i-1} \tag{3.43}$$
$$\dot{\boldsymbol{p}}_i = \dot{\boldsymbol{p}}_{i-1} + \dot{d}_i \boldsymbol{z}_{i-1} + \boldsymbol{\omega}_i \times \boldsymbol{r}_{i-1,i} \tag{3.44}$$

Jacobian Calculation

In equation 3.32, we introduced the Jacobian as a mapping between cartesian space and joint space. As we have seen in the former examination, the Jacobian maps rotational and translational velocity towards cartesian velocity. Thus, we can formulate the Jacobian as such:

$$\boldsymbol{J} = \begin{pmatrix} \boldsymbol{J}_p \\ \boldsymbol{J}_r \end{pmatrix} = \begin{pmatrix} \boldsymbol{J}_{p_1} & \dots & \boldsymbol{J}_{p_n} \\ \boldsymbol{J}_{r_1} & \dots & \boldsymbol{J}_{r_n} \end{pmatrix} \quad , \textit{with } \boldsymbol{J} \in \mathbb{R}^{6\times n} \tag{3.45}$$

where each $\boldsymbol{J}_{p_i}$,$\boldsymbol{J}_{r_i}$ describes a 3-dimensional vector, which relates each joint velocity $\dot{q}_i$ to the linear velocity. $\boldsymbol{J}_{p_i}$ maps x, y, z components of translational velocity into cartesian space. Equally $\boldsymbol{J}_{r_i}$ relates the orientation in terms of Euler angles R, P, Y.

$$\boldsymbol{J}_{p_i} = \begin{bmatrix} J_{p_{i,x}} \\ J_{p_{i,y}} \\ J_{p_{i,z}} \end{bmatrix} \qquad \boldsymbol{J}_{r_i} = \begin{bmatrix} J_{r_{i,r}} \\ J_{r_{i,p}} \\ J_{r_{i,y}} \end{bmatrix} \tag{3.46}$$

In particular the mapping function looks like the following:

$$\begin{bmatrix} \dot{p}_x \\ \dot{p}_x \\ \dot{p}_x \end{bmatrix} = \boldsymbol{J}_p \dot{\boldsymbol{q}} \qquad \begin{bmatrix} \dot{p}_r \\ \dot{p}_p \\ \dot{p}_y \end{bmatrix} = \boldsymbol{J}_r \dot{\boldsymbol{q}} \tag{3.47}$$

We can now compute the full Jacobian, taking again the different types of joints into account. We compute each pair of $\boldsymbol{J}_{p_1}$,$\boldsymbol{J}_{r_1}$ in equation 3.45 according to:

$$\boldsymbol{J} = \begin{pmatrix} \boldsymbol{J}_{p_1} \\ \boldsymbol{J}_{r_1} \end{pmatrix} = \begin{cases} \begin{pmatrix} \boldsymbol{z}_{i-1} \times (\boldsymbol{p} - \boldsymbol{p}_{i-1}) \\ \boldsymbol{z}_{i-1} \end{pmatrix} & \textit{for revolute joints} \\ \\ \begin{pmatrix} \boldsymbol{z}_{i-1} \\ \boldsymbol{0} \end{pmatrix} & \textit{for prismatic joints} \end{cases} \tag{3.48}$$

For a detailed and well explained derivation of the above computation, the interested reader is referred to [45] and [49].

3.3.2. Jacobian Calculation of Moving Points

In the previous section, calculated the Jacobian with respect to all related joints along one kinematic chain ending with the tool joint at the tip. Within this calculation, the assumption is taken that all joint position are at the end point of the link. The axis of movement is Z.
For the purpose of collision avoidance based on a closest point pair, we need the Jacobian in respect of the closest point p_1, which varies from the center point on the z-axis. As depicted in figure 3.9, we need to find a transformation from the Jacobian calculated for frame origin O_i to frame origin O_{cp}. This means, we need a mapping from the joint velocities to the linear and angular velocities at point O_{cp}. Since the closest point is part of the

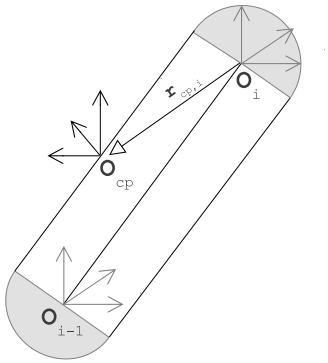


Figure 3.9.: Jacobian has to be calculated in respect to frame O_{cp}. The coordinate frame transformation can be realized with a twist, given the relative transformation $r_{cp,i}$ between O_i and O_{cp}.

capsule, for which origin we can easily calculate the Jacobian, we can treat this as a static transformation. Thus, we can state the following relation:

$$\begin{aligned} \dot{\boldsymbol{p}}_{\boldsymbol{cp}} &= \dot{\boldsymbol{p}}_{\boldsymbol{i}} + \boldsymbol{\omega}_{\boldsymbol{i}} \times \boldsymbol{r}_{cp,i} \\ \boldsymbol{\omega}_{\boldsymbol{cp}} &= \boldsymbol{\omega}_{\boldsymbol{i}} \end{aligned} \tag{3.49}$$

By utilizing the skew-symmetric operator $S(\bullet)$, we can write the above relation in matrix notation.

$$\begin{pmatrix} \dot{\boldsymbol{p}}_{\boldsymbol{cp}} \\ \boldsymbol{\omega}_{\boldsymbol{cp}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{I} & -\boldsymbol{S}(\boldsymbol{r}_{cp,i}) \\ \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \dot{\boldsymbol{p}}_{\boldsymbol{i}} \\ \boldsymbol{\omega}_{\boldsymbol{i}} \end{pmatrix} \tag{3.50}$$

This results in a twist computation as the linear projection now describes a $\mathbb{R}^{6\times 6}$ matrix. We can use this formulation in order to obtain the Jacobian at point O_{cp}. If we substitute $\dot{\boldsymbol{p}}_{\boldsymbol{cp}}$ and $\boldsymbol{\omega}_{\boldsymbol{cp}}$ with the Jacobian mapping function, equation 3.50 yields

$$\begin{aligned} \begin{pmatrix} \dot{\boldsymbol{p}}_{\boldsymbol{cp}} \\ \boldsymbol{\omega}_{\boldsymbol{cp}} \end{pmatrix} &= \begin{pmatrix} \boldsymbol{I} & -\boldsymbol{S}(\boldsymbol{r}_{cp,i}) \\ \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \dot{\boldsymbol{p}}_{\boldsymbol{i}} \\ \boldsymbol{\omega}_{\boldsymbol{i}} \end{pmatrix} \\ \boldsymbol{J}(\boldsymbol{p}_{cp}, \boldsymbol{q})\dot{\boldsymbol{q}} &= \begin{pmatrix} \boldsymbol{I} & -\boldsymbol{S}(\boldsymbol{r}_{cp,i}) \\ \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \boldsymbol{J}(\boldsymbol{p}_i, \boldsymbol{q})\dot{\boldsymbol{q}} \\ \boldsymbol{J}(\boldsymbol{p}_{cp}, \boldsymbol{q}) &= \begin{pmatrix} \boldsymbol{I} & -\boldsymbol{S}(\boldsymbol{r}_{cp,i}) \\ \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \boldsymbol{J}(\boldsymbol{p}_i, \boldsymbol{q}) \end{aligned} \tag{3.51}$$

With the above formulation, we can calculate the Jacobian at any point of the capsule. Most kinematic or dynamic libraries provide already the capability of computing the Jacobian along the kinematic chain up to the end effector position. Thus, for every update of the solver, we have to compute the relative transformation $\boldsymbol{r}_{cp,i}$ and obtain the according Jacobian based on equation 3.51.

Part II.

Implementation Details

CHAPTER 4

IMPLEMENTATION ENVIRONMENT

In this chapter, we introduce the environment used for the implementation of this work. Hereby, we provide conceptional overview about the underlying real time critical base implementation, namely ROS_control, which serves as a generic interface between robot control and respective hardware. At this point, we assume the reader to be familiar with ROS. A good introduction, tutorials and detailed information can be found in [39] and the official website[1].

Furthermore, we introduce the SoT as a framework in terms of implementing tasks and dynamically push and pop them on runtime. Hereby, we describe the Dynamic Graph (DG). This serves as a computational graph and each task can be seen as one entity inside this graph.

Latest, the closest points are calculated utilizing an open-source library called Fast Collision Library (FCL), which brings already mechanisms for calculating proximities. During this work, the capabilities of this library is extended for capsules.

4.1. ROS_control

In a wide sense, ROS_control describes a combination of packages for building robot agnostic controllers. It provides interfaces for controllers and various hardware systems to generically separate the controllers from the underlying hardware. ROS_control thus can be seen as a hardware abstraction layer (HAL). On the upper side, ROS_control enables a generic ROS compatible programming interface, which allows hardware agnostic compu-

[1] http://www.ros.org/

tation of various application algorithms. On the lower side, it communicates in a real time safe environment with the concrete hardware. This mechanism is monitored by a independent observer, called the controller manager, which allows to load, switch and unload multiple controllers on runtime. More information can be found in [40].

During the development of this work, we implemented a so-called SoT-Controller and integrated it into ROS_control. This integration implies a seamless migration and execution on the real robot hardware through the implemented HAL for REEMs hardware. To overcome any real time constraints inside the computation of the stack, we exposed this computation into a separate thread. This not just detaches the SoT from any hard timing condition, but rather allows us to modify the update frequency inside the SoT independently from ROS_control. At each `update` function call of the SoT-Controller, it fetches the latest result q, representing the accumulated joint states. Those joint state values are passed through a `joint position controller` and finally to the robots motor controllers. This abstract description is illustrated in figure 4.1.

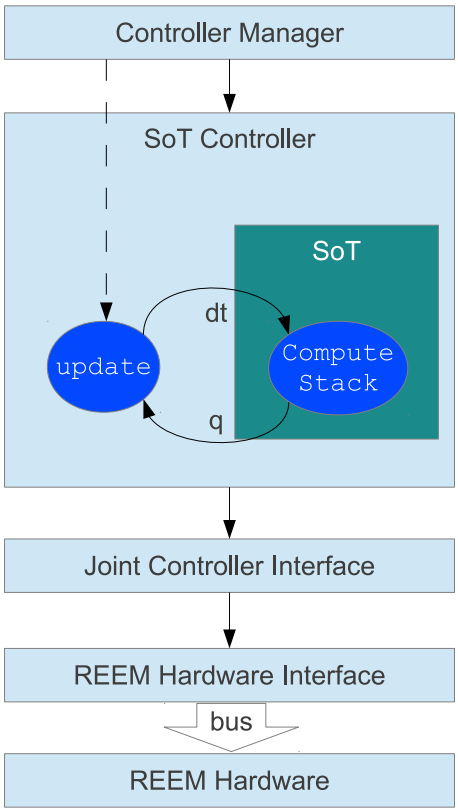


Figure 4.1.: Abstract overview of the SoT-controller architecture inside ROS_control. The controller manager cyclically calls the `update` function for all subscribed controllers. This update function fetches the latest result from the independently running SoT. The joint states are getting passed over various interfaces finally to the communication bus and the respective motor control boards.

4.2. Stack of Task - Framework

During the theoretical part in chapter 2, we introduce the SoT as a GIK solver and provide mathematical formulations on how to define a task inside this solver. In this section, we consider the SoT from a software framework point of view. With this being said, in the remainder of this section, the use of the abbreviation SoT refers to the framework rather than the hierarchical solver.

The SoT comprises mainly two key functionalities.

- Dynamic computational graph connecting entities through input/output signals
- Hierarchical task solver (called sot-core) realized as entities inside the dynamic graph

4.2.1. Dynamic Graph

The dynamic graph (DG) provides a computational graph, where entities are connected through input and output signals [38]. Hereby, every output signal is combined with an internal function of the entity, gathering all necessary information. This happens by consuming the input signals and produce the result on the output signal. Output signals from one entity can be plugged as an input signal of any other entity.

For every cycle of the control update loop, the DG gets recomputed based on backtracking all linked entities. Triggering an output signal executes the assigned output function inside the entity. Equally, this function might consume linked input signals. Since these input signals represent output signals of other entities, those entities are computed as well. Let us consider the DG illustrated in figure 4.2. In order to calculate the output value of Entity E, all precedent entities are backtracked until the initial values c_1 and c_2.

Depending on the size of the DG, this might result in a big overhead of recomputation, even though some entities do not provide new values. In the shown graph, we plug two constant values c_1 and c_2. Computing now the value of `Entity E` would backtrack the complete graph with no new result, as the initial values are constant. In order to prevent this overhead and recalculate only outdated entities, all signals are time dependent. This timestamp allows a recomputation of the entity only if the according input signals are outdated.

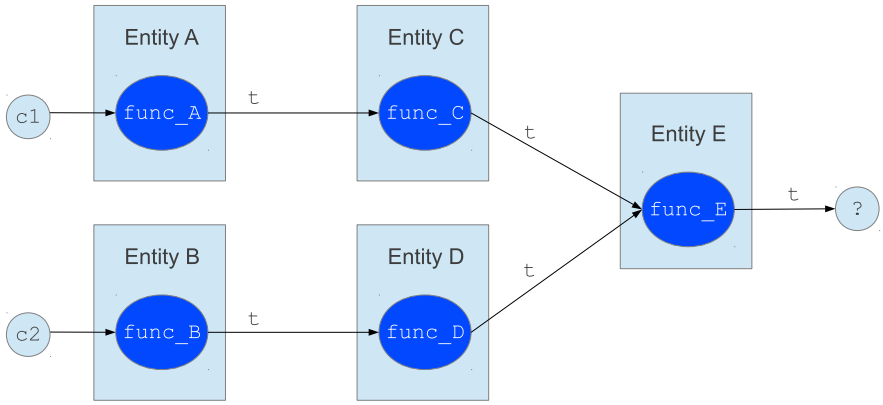


Figure 4.2.: Dynamic Graph with five entities $A - E$. Calculating the output of `Entity E` triggers a recursive backtracking until all entities are updated. In order to prevent an unnecessary recalculation of unchanged entities, signals are computed with a timestamp t. This timestamp keeps track of changes, which indicates whether an entity is outdated and has to be recomputed.

An example would be an entity called *feature*, which calculates the error between the actual and desired position of the end-effector position. Thus, this entity comprises two input signals $p(t)$ and $p_{des}(t)$. The respective output signal provides the error, hence the name $e(t)$. The function linked to the output signal obviously computes the difference between p and p_{des} at time t_i. Algorithm 1 describes this simple function, where the two input signals are fetched at the same timestamp and the resulting error is computed. The output signal equals the return value of the function.

Algorithm 1: error computation in entity feature

/* signals have to be initialized before */
input : Timestamp t
output: Error e
p = `p_signal_in(`t`)` ;
p_des = `p_des_signal_in(`t`)` ;
e = p_des - p ;
return e ;

4.2.2. SoT Core

As mentioned in the beginning of this chapter, the SoT (meaning the hierarchical solver in this case) is realized as an dedicated entity inside the DG. This entity provides functionalities to dynamically push and pop task entities.

We described already in the theoretical part, how a task definition looks like. Since we minimize a QP in the form of

$$\begin{aligned} \boldsymbol{Ax} - \boldsymbol{b} \quad & \textit{generic form} \\ \boldsymbol{J\dot{q}} - \boldsymbol{\dot{e}} \quad & \textit{inverse kinematic} \end{aligned}$$

We have to specify the Jacobian $\boldsymbol{J}$ as well as the residual $\dot{e}$, which represents - in this case - the error between actual and desired position. The interface for a task implementation thus provides two output signals for the Jacobian and the error, respectively.

In the previous section, we explained how output signals are handled inside the dynamic graph. The assigned output function for each signal is task dependent. An example positioning task for an end-effector can then easily be formulated by using the error calculation of the feature entity (see algorithm 1) and the Jacobian for the end-effector.

We have to point out that at this point, we have only designed a task. We are not solving this task though. The task becomes part of the solver by explicitly pushing it onto the stack. The hierarchical solver in itself describes another entity, which holds all tasks, currently pushed on the stack. This entity solves the stack by fetching all `jacobian` and `error` output signals from all tasks. This will be transformed into one big, hierarchy respecting Jacobian and error. This will be finally solved and the result is provided as an output signal. We are not going any deeper into details here. The main information should be the requirements for designing tasks and push them on the stack for getting considered by the solver. Mathematical details can be found in [11][13]. An deeper insight into the development of the framework is provided in [37].

Furthermore, we want to remind that the mathematical solver indeed solves the generic form of the above stated formulations. This being said, there is no need to specify $\boldsymbol{A}$ exclusively as a Jacobian matrix. This allows us also to specify tasks directly in joint space.

$$\begin{aligned} \boldsymbol{Ax} - \boldsymbol{b} \quad & \textit{generic form} \\ \boldsymbol{I\dot{q}} - \boldsymbol{\dot{q}_d} \quad & \textit{joint space task} \end{aligned}$$

Where $\boldsymbol{I}$ denotes an identity matrix. The residual therefore has to be given in joint space, hence the naming $\boldsymbol{q_d}$, which specifies a desired joint velocity.

4.3. Fast Collision Library (FCL)

FCL is a fully templated library for generic collision detection and proximity calculations [43]. It is designed to allow hierarchical proximity queries on various collision types among others such as axis-aligned bounding box (AABB), oriented bounding box (OBB) and swept sphere volume (SSV). It provides implementations based on Gilbert-Johnson-Keerthi (GJK) [57] [20] algorithms for proximity as well as collision detection for many geometry primitives such as boxes, cubes and spheres.

Among them, a concrete capsule proximity implementation was not available during the time of this development. Based on the fact, that FCL is completely open source and developed with the standard template library (STL), it was possible to integrate the capsule decomposition into FCL. The work presented in chapter 3.3 is at the time of writing integrated in FCL and can be obtained on github[2].

The use of FCL as the main proximity library allows us to not exclusively focus on a hard-coded capsule decomposition. This being said, the same code can be used for example when certain body parts are wrapped with cubes as collision geometries. Contrary, as FCL is part of the core inside frameworks such as MoveIt!, the same capsule decomposition can be used for other applications.

[2] https://github.com/flexible-collision-library/fcl

CHAPTER 5

COLLISION AVOIDANCE IMPLEMENTATION

In chapter 3, we analyze the theoretical aspects of our approach for collision avoidance. We examine in section 3.1 how a capsule decomposition and the according closest points between two capsules are sufficiently convex enough to cover the robots collision model. Additionally, we introduce a velocity damping method in 3.2, which restricts any positive velocity component along the directional vector towards the possible collision center.

In the previous chapter 4, we briefly introduced the SoT as a framework and FCL as the according library for proximity queries. This chapter furnishes insights about the implementation of the closest point calculation and velocity damping as entities inside the DG.

Although it can be seen that those two components alone perform well to protect any (self-)collision, in practice there is a need for additional components. There are components needed to operate the robot with a smooth transition of movements. We additionally introduce task definitions for joint velocity and joint position limits as well as a weighting task, which can be used to operate the robot with the necessary safety.

In the following, we have to carefully differentiate the wording between an *entity* and a *task*. Technically speaking, a task denotes an entity inside the DG. Contrary, an entity is not a task, thus it is not able to be pushed on the stack and be considered by the solver.

5.1. FCL Entity

As the name might reveal, the FCL entity is responsible for querying proximities based on FCL. We mentioned in the previous section, that during the development of this work, we enhanced FCL with the capsule capabilities. Thus, on startup, we load the capsule parameters such as origin, length and radius from a robot description file[1]. This parses theses information into a capsule collision object and provides a full collision matrix of all given capsules.

The collision matrix is represented by output signals for each collision pair. Each output signal provides the closest point from one capsule to another. Hereby, the signals are ordered according to the naming convention `<capsule-link-1><capsule-link-2>`, which indicates the closest point on the first capsule towards the closest point on the second capsule. Table 5.1 gives an example for three capsules, namely the right arm, the left arm and the torso. The table shows the resulting output signals. Inside the concrete application, the names are according to the names given in the robot description file.

	right-capsule	left-capsule	torso-capsule
right-capsule	X	`<right-capsule left-capsule>`	`<right-capsule torso-capsule>`
left-capsule	`<left-capsule right-capsule>`	X	`<left-capsule torso-capsule>`
torso-capsule	`<torso-capsule right-capsule>`	`<torso-capsule left-capsule>`	X

Table 5.1.: Collision Matrix and the produced output signal names.

This full feature collision matrix might become relatively large, depending on the amount of capsules. This might lead to a large computational cost. At this point however, we only provide output signals inside the DG. With this in mind, the actual proximity computation only gets triggered, when an update of the respective signal is requested by another entity.

[1] Unified Robot Description Format(URDF), see `http://wiki.ros.org/urdf`

5.2. Velocity Damping Task

The implementation of the velocity damping inside a task follows after the FCL entity. We pointed out in the previous chapter 4, that for defining a task inside the stack, we have to implement a function for the `Jacobian` and `error` signal. For convenience, we repeat the inequality formulation of the velocity damping constraint in equation 3.24 here again.

$$\boldsymbol{n}^T \boldsymbol{J}(\boldsymbol{p_1}, \boldsymbol{q})\dot{\boldsymbol{q}} \geq -\epsilon \frac{d - ds}{\Delta t} \tag{5.1}$$

In respect of the above equation, we implement the left hand side of the inequality inside the `Jacobian` function. The right hand side, the velocity boundary, is implemented in the `error` function. Moreover, we have to specify a set of parameters as input. The equation requires explicitly $\boldsymbol{p_1}$, d_s and ϵ as given, implicitly it also requires p_2 to be set for calculating the remaining values such as d and $\boldsymbol{n}$.

We will see during the examination of the experiments in chapter 6, that d_s and ϵ have to be set as constant input signals as they heavily depend on the robot setup. The two points $\boldsymbol{p_1}, \boldsymbol{p_2}$ can be obtained from the FCL entity, as these points are exactly the closest point pair of two collision objects. The connection will be done by plugging the output signals of the FCL entity, into the $\boldsymbol{p_1}, \boldsymbol{p_2}$ input signals of the velocity damping task. The Jacobian has to be calculated according to the current joint positions. Our implementation takes the Jacobian with respect to the link origin. This calculation is most often already implemented in various kinematic or dynamic libraries; also inside the SoT. In order to get a Jacobian with respect to the closest point, we apply a twist transformation as explained in section 3.3.

The DG setup for the self-collision avoidance is illustrated in the following graphic 5.1.

We execute only one velocity damping task for avoiding all possible self-collisions. This means, that all collision pairs are on the same level inside the hierarchy. Thus, we configure the task velocity damping to possess multiple input sockets for all closest point pairs and the according control parameters such as d_s and ϵ. This formulation treats all collisions with the same priority.

On the same hand, we only plug the collision pairs we really need to protect the complete robot. This implies an exclusion of adjacent body parts, since they are constantly in collision and would thus violate the solver during the complete execution time. Furthermore, we can exclude collision pairs, which are not able to collide based on the mechanical construction of the robot, such as left and right shoulder. This makes the configuration effort rather simple and the computational complexity is on a minimal.

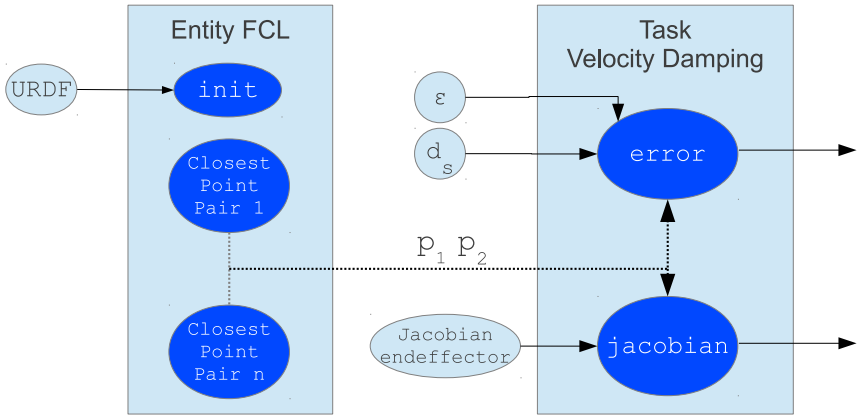


Figure 5.1.: Dynamic Graph setup for the self-collision avoidance. EntityFCL provides the closest point pair calculation. These points are plugged into the velocity damping task for calculating $\boldsymbol{n}$ and d. Values such as d_s and ϵ are provided as constant signals. Most kinematic or dynamic libraries provide already ways to compute a Jacobian with respect to the end-effector. We provide the end-effector Jacobian as an input signal and apply a twist operation in order to obtain the Jacobian with respect to the closest point. The output signals of the velocity damping are in accordance with the abstract task definition of the SoT, since it requires a `Jacobian` and `error` function.

5.3. Joint Limits Task

The two entities, we described previously, are capable of preventing any self-collision with the robot. However, if we execute only the velocity damping as an exclusive task, we run into trouble as the joint limits might get exceeded. This results in heavy discontinuities which might in the end even destroy the robot. For this purpose of protecting the motors and gears, we introduce two tasks in joint space, which take care about joint position exceeding as well as joint velocity limits. In the generic robot description file, we can calibrate persistent values for the minimal and maximal position for each joint.

The hierarchical solver inside the SoT operates inside the velocity domain. Thus, for a positioning task, we have to specify a Jacobian matrix, which translates joint space velocities into cartesian space velocities. This formulation allows us to specify the error residual, for example the error between desired and actual position of an end-effector, in cartesian space.
At this point, we want to specify a joint space residual, namely the distance between the current joint position and the maximal or minimal boundary. In order to specify this in a similar way, we replace the Jacobian matrix by a simple identity matrix. The task definition then becomes the following:

$$\boldsymbol{I}\dot{\boldsymbol{q}} \geq \dot{\boldsymbol{q}}_{\boldsymbol{err}} \quad \textit{with } \dot{\boldsymbol{q}}, \dot{\boldsymbol{q}_{\boldsymbol{err}}} \in \mathbb{R}^{1\times n} \tag{5.2}$$

where $\boldsymbol{I}$ denotes the identity matrix and $\dot{\boldsymbol{q}_{\boldsymbol{err}}}$ the error in joint velocity. We have to note

here, that the error has to be given in the velocity domain, rather than position. This implies, we have to take the first derivative of the joint position error. Thus the above formulation turns into

$$\boldsymbol{I}\dot{\boldsymbol{q}} \leq \frac{\boldsymbol{q_{max}} - \boldsymbol{q}}{\Delta t} \quad \textit{with } \boldsymbol{q_{max}}, \dot{\boldsymbol{q}} \in \mathbb{R}^{1\times n} \tag{5.3}$$

where $\boldsymbol{q_{max}}$ specifies the maximal joint position. Δt denotes the time between two computation cycles, which we use to compute a first numerical derivative between two joint positions in time. We enhance this formulation with an additional control gain k to ease any recalibration on runtime. The final equation formulation is presented in equation 5.4

$$\begin{aligned} \boldsymbol{I}\dot{\boldsymbol{q}} &\leq \frac{\boldsymbol{q_{max}} - \boldsymbol{q}}{k\Delta t} \\ \boldsymbol{I}\dot{\boldsymbol{q}} &\geq \frac{\boldsymbol{q_{min}} - \boldsymbol{q}}{k\Delta t} \end{aligned} \tag{5.4}$$

We have to point out, that k is placed in the denominator. This has the meaning, that an increase of k leads to an decrease of maximal joint position changes. We implement the joint position limits as a double bounded task, restricting the joint position in the lower and upper boundary.

The above formulation prevents the solver to resolve any joint state vector $\boldsymbol{q}$, which would exceed any joint positions. However, it does not take the joint velocity into considerations. Image equation 5.4 with a low value for k and a current joint position vector $\boldsymbol{q}$, which is in the middle of the position limits. There can be situations, where the resulting allowed velocity $\dot{\boldsymbol{q}}$ might be too high for the motors. On the same hand, restricting the velocity with a rather big value for k also shrinks down low velocities.

Based on the above considerations, we implement a velocity clamping on top of it. Hereby, in the robot description file, we further specify a maximal joint velocity value. With this, we execute the joint limits with the above formulation, however check if the resulting boundary, the right hand side of the inequality, exceeds the maximal velocity. The task definition looks like the following:

$$\begin{aligned} \boldsymbol{I}\dot{\boldsymbol{q}} &\leq \begin{cases} \frac{\boldsymbol{q_{max}}-\boldsymbol{q}}{\Delta t} & \text{if } \dot{\boldsymbol{q_{err}}} \leq \dot{\boldsymbol{q}}_{\boldsymbol{max}} \\ \dot{\boldsymbol{q}}_{\boldsymbol{max}} & \text{else} \end{cases} \\ \boldsymbol{I}\dot{\boldsymbol{q}} &\geq \begin{cases} \frac{\boldsymbol{q_{min}}-\boldsymbol{q}}{\Delta t} & \text{if } \dot{\boldsymbol{q_{err}}} \leq \dot{\boldsymbol{q}}_{\boldsymbol{min}} \\ \dot{\boldsymbol{q}}_{\boldsymbol{min}} & \text{else} \end{cases} \end{aligned} \tag{5.5}$$

5.4. Joint Weighting Task

Finally, we introduce a task, which restricts the movement of expensive joints such as the torso. This task is not necessarily a requirement for safe operations with the robot. It is more a task, which limits the movement of heavy torques and inertia. It further improves the workspace of the robot, since a weighting of the torso means a lower movement of the most commonly shared part of the kinematic chain. The torso will be shared between the head and both arms. However, those tasks of positioning head and arms might be in different priority level such that the torso almost exclusively respects the highest of those tasks. We can correct this by weighting the torso, which is getting compensated by foster a larger movement of different joints, e.g. in the respective arms.

We can easily formulate this task again in joint space. In contrast to the pure identity matrix, we multiply a diagonal weighting matrix with entries according to the inertia of the robot. As we try to minimize the movement as much as possible, we set the resulting velocity to zero.

$$\boldsymbol{W}\boldsymbol{I}\dot{\boldsymbol{q}} = 0 \quad \text{with } \boldsymbol{W} \in \mathbb{R}^{n \times n} \tag{5.6}$$

The equation successfully limits the movement according to the value set on the main diagonal of the weighting matrix. However, when we look at this formulation inside the stack, this approach has some limitations. When we set this task as a rather high priority, the robot does not move. It is clear, that when this task will be fully solved, all DoF are getting used and there is no nullspace for lower priority tasks. With this in mind, we have to set this weighting task always at the least priority position, in order to minimize the velocities. The optimization is done according to the weights as much as possible, yet not restricting any other tasks.

CHAPTER 6

EXPERIMENTS

The solution of this work was validated with a couple of experiments and stress tests. We show in the following, that this approach provides a safe way to avoid collisions with the robot itself as well as avoiding external obstacles. Reviewing the goal statements of the introduction 1.2, we conducted the following experiments to get a quantitative evaluation:

- Providing a piecewise continuous trajectory of closest point pairs
- Avoiding collisions between the actuated hand and external obstacles
- Avoiding full self-collision

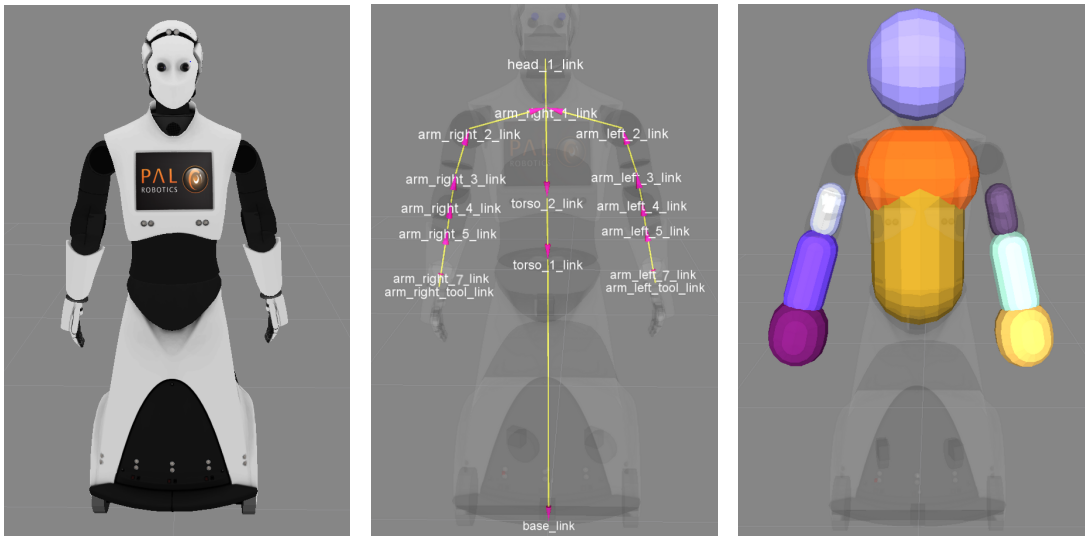
6.1. Robot Hardware Setup

The conducted experiments have been mainly examined with a real time physics simulator, namely Gazebo. It simulates an appropriate robot behavior in terms of joint position as well as joint velocities.

The full body self-collision avoidance is running successfully on the robot. This being said, this chapter briefly introduces the hardware on the robot (REEM-H in this case) to give the reader an overview over the experiment circumstances.

REEM-H comprises an embedded computer, running a Ubuntu 12.04 LTS version with a real time kernel. The specifications as they are provided on PAL Robotics website at the time of writing are listed in table 6.1.

Furthermore, the used robot model is presented below. The three images denote the internals of the robot (figure 6.1). In the first image, we can see the visual model of REEM-H. The kinematic chain is illustrated in the second image. Hereby, the names of the used links are shown as well [1]. In the remainder of this chapter, we may refer to them, as we calculate the closest point pair based on those links. Finally, the third image gives an idea about the undertaken capsule decomposition.



(a) REEM-H Model (b) Kinematic Chains (c) Capsule Decomposition

Figure 6.1.: REEM-H model explained: a) visual surface model. b) kinematic chain of all links. These link names are used as a reference in the remainder of this chapter. c) implemented capsule decomposition. Three capsules are placed on each arm, two on the upper body as well as one for the head.

During the development of this work, we could determine the shown capsules in figure 6.1(c) as being sufficiently necessary to protect all possible self-collisions. Since we are implementing a safety distance threshold on every capsule, the resulting safety padding of all capsules sufficiently covers the left out body parts. Table 6.1 lists the capsule decomposition of the body parts.

6.2. Experiments

All of the following concrete experiments comprise two major tasks. The highest prioritized task of all experiments is joint limit avoidance. This is followed by the task velocity damping introduced in chapter 3.2, which denotes the (self-)collision avoidance task.

[1] For completeness: We omitted arm_link_6 on both arms as they are located in the same position than the tool link. The graphic looks clearer this way and link_6 carries no necessary need in terms of capsule decomposition.

REEM-H specification	
weight	90 KG
height	1.70 meter
DOF	22
Computer	Intel Core2Duo
OS	Ubuntu 12.04 LTS
Kenel	Xenomai real time toolchain

Table 6.1.: Hardware specification of REEM-H

Capsule Decomposition	
Right Arm	arm_right_3_link
	arm_right_5_link
	arm_right_7_link
Left Arm	arm_left_3_link
	arm_left_5_link
	arm_left_7_link
Upper Body	torso_1_link
	torso_2_link
Head	head_1_link

Table 6.2.: Capsule Decomposition of REEM-H

Hereby, this is configured according to the respective exercise (such as external collision avoidance or self-collision avoidance). Lastly, the active task for positioning the tool center of the left as well as the right arm is placed at the bottom of the stack. Figure 6.2 depicts the basic stack used for these experiments. During the experiments, we change the order of the left and right wrist, however they are constantly placed at the bottom of the task.

Figure 6.2.: Basic hierarchy stack of used tasks for the experiments. As illustrated, joint limits have always the highest priority. The collision avoidance task takes precedence over the tasks of interest such as the right or left wrist. In the conducted experiments, we will also examine the behavior of changing the order of left and right, yet constantly placing them at least priority.

Beforehand, we implemented two strategies for integrating the SoT inside ROS_control of the REEM robots. Xenomai respectively ROS_control operates with a frequency of $100Hz$. This means, that the physical update cycle is lower bounded to $100Hz$ and the computation of the stack has to produce a result with in this cycle. On the same hand, we integrated the SoT as a asynchronous, not real time safe thread, next to ROS_control. With this decoupling, we can prevent any real time violations to be visible on the robot. This further implies, that we can achieve a higher cycle rate exclusively for the SoT, providing the current solution for every cycle of ROS_control. The purpose of these different cycle rates is

essentially the capability of shrinking the Δt inside the update cycle of the SoT. We recall the task definition in 2.26 again:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{J}(\boldsymbol{q}, \boldsymbol{t})\dot{\boldsymbol{q}} \\ \dot{\boldsymbol{x}} &= \boldsymbol{J}(\boldsymbol{q}, \boldsymbol{p})\frac{\Delta \boldsymbol{q}}{\Delta t} \end{aligned} \tag{6.1}$$

From equation 6.1, it is clear that lowering Δt yields to smoother results in $\dot{\boldsymbol{x}}$ and thus to a smoother trajectory on the robot. All experiments, which are presented below, were executed with a rate of $100Hz$ inside the SoT, because of limits in computation power of the embedded computer inside REEM.

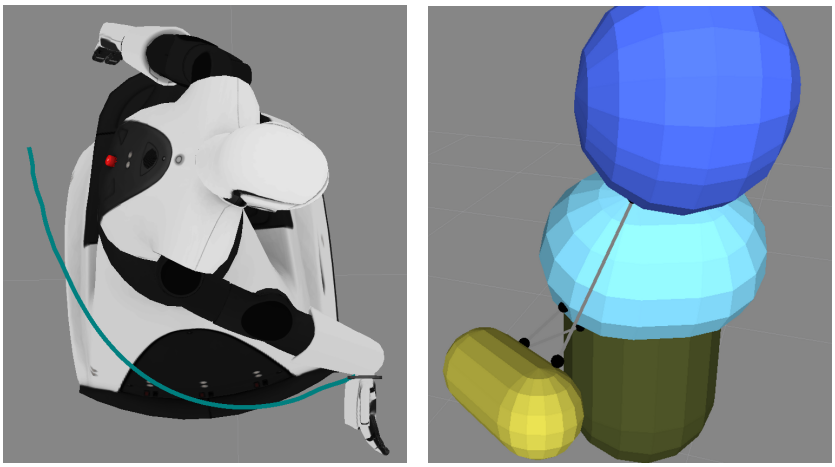
6.2.1. Continuous Trajectory of Closest Points

In chapter 3.1, we examined the benefit of exercising a capsule decomposition over a mesh collision geometry. The goal in this experiment is to demonstrate the required continuity of the closest point pair. Informally spoken, monitoring the trajectory of each point of the according point pair must result in one single path and is not allowed to jump in position.

For this experiment, we move the right arm from the right ($x : -0.2m, y : -0.3m, z : 1.1m$) to the left($x : 0.2m, y : 0.3m, z : 1.1m$). Hereby, we examine the development of the closest point pair between arm_right_5_link and the Upper Body as well as Head. The arm movement as well as the closest point calculation is shown in the two figures below (figure 6.3). Figure 6.3(b) contains information about the closest point pairs at time t_k for a specific k. The lines mark the connection between the two points which form a collision pair.

The result of this particular experiment is exposed below. We can see in the results (figure 6.4), that the requirement is fulfilled as the trajectory for all closest points is continuous. The plots show the path in x and y, since the executed movement is planer in $z : 1.1m$. The left plot describes the points, which are attached at the `arm_right_5_link`. Their data is expressed with respect to the global base frame. Equally, the right plot represents the points on the Upper Body and Head.

This experiment was successfully executed with all possible collision pairs with similar results. We encounter a rather negative result, when joint limits or clamping situations occur as the resulting joint velocities might increase dramatically. This inevitably leads to a high displacement of the closest points.



(a) Executed trajectory (b) Closest Point Pairs

Figure 6.3.: Closest Point Trajectory: a) shows the trajectory the right arm executed. The robot moved around the torso from the right ($x : -0.2m, y : -0.3m, z : 1.1m$) to the left($x : 0.2m, y : 0.3m, z : 1.1m$) (seen from the robot perspective). b) denotes the closest point pairs between the three collision pairs at t_k. The development of this calculation has to be continuous for all t_i. The lines are providing info which points belong to which collision pair.

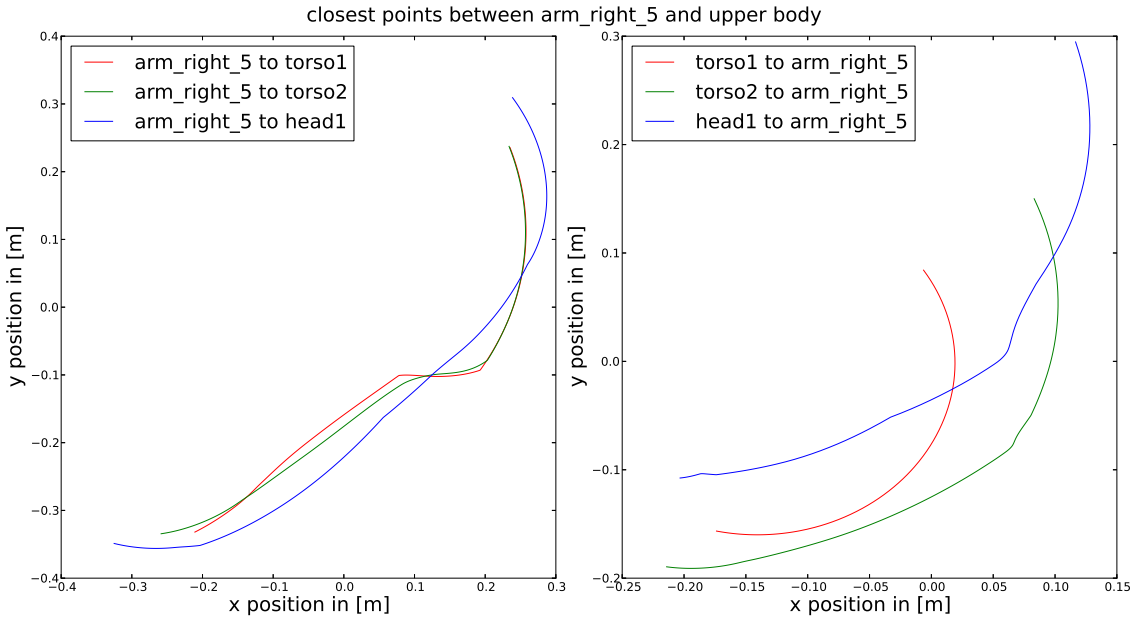


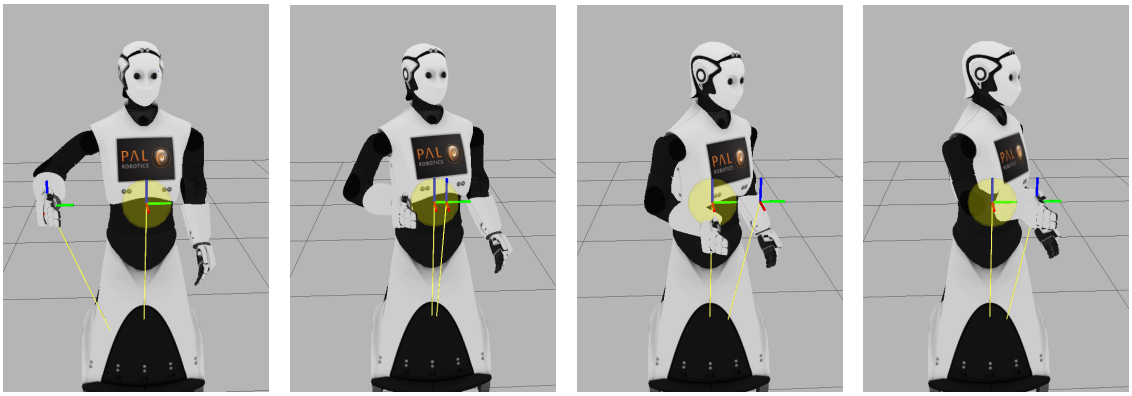
Figure 6.4.: closest point development between `arm_right_5_link` and Upper Body (consisting of `torso_1`, `torso_2`, `head_1`). Left diagram shows the closest points lying on the capsule of the right arm towards the upper body. Similarly, the right diagrams displays the counter points on the upper body, which are pointing towards the right arm.

6.2.2. External Collision Avoidance

To begin with, we conduct the experiment of avoiding external obstacles coming across a predefined trajectory of one of the arms. Hereby, we specify a trajectory input signal inside the dynamic graph, which provides the desired position of the right tool joint. Equally, we specify a similar signal representing the obstacle position. Thus, we have to configure the closest point pair p_1 as the right wrist signal and the p_2 as the obstacle signal.

Static Object vs. Moving Arm

The first experiment describes a statically placed collision obstacle. The right wrist has to avoid this obstacle, yet minimizing the error between the actual and desired position to the best. The task specifying the left wrist position is not on the stack in this experiment. The predefined trajectory denotes a path from the left to the right and is shown in the following sequence of screen shots 6.5. The control gain ϵ for the velocity damping task is set to 1, d_s is set to $0.1m$. The control gain k for the right wrist task is set to 10. In the sequence of



(a) (b) (c) (d)

Figure 6.5.: Sequence of screen shots for the predefined trajectory. The yellow ball represents the external obstacle, which has to be avoided. a) starting position b) collision avoidance gets activated as the obstacle boundary is reached c) alternate trajectory trying to minimize the error d) reaching desired position and collision avoidance is deactivated.

screen shots, we can see that the arm is following the trajectory until the collision object is reached. At this point the collision avoidance task becomes part of the active set and the movement along the directional vector is limited. In order to minimize the error, an alternative path is solved and the obstacle is avoided. A closer look at the trajectory is provided in figure 6.6. It is remarkable, that the solver computes a different solution for the way back. Reasons for this might be the different starting configuration, where the HCOD fosters different joint movements at different times.

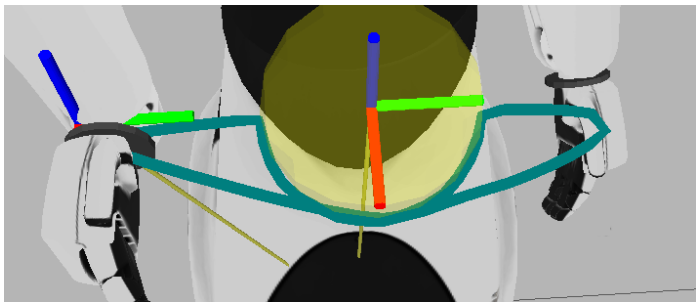


Figure 6.6.: Avoidance of external obstacle. In order to avoid the obstacle, yet minimizing the error towards the goal position, the movement yields along the collision object. The trajectory computed for the way back differs from the initial way. Reason for this are different starting positions and conditions. The HCOD solves the resulting equation system differently and fosters joint movements, which vary from the initial solution.

The distance over time is depicted in the plot below (Figure 6.7). Within this, we can see that the critical distance threshold d_s is set to $0.1m$, which will not be violated. The distance between the obstacle and the wrist stays constantly over this specific safety zone. When we look at the position trajectory in figure 6.8, we can also see the expected result.

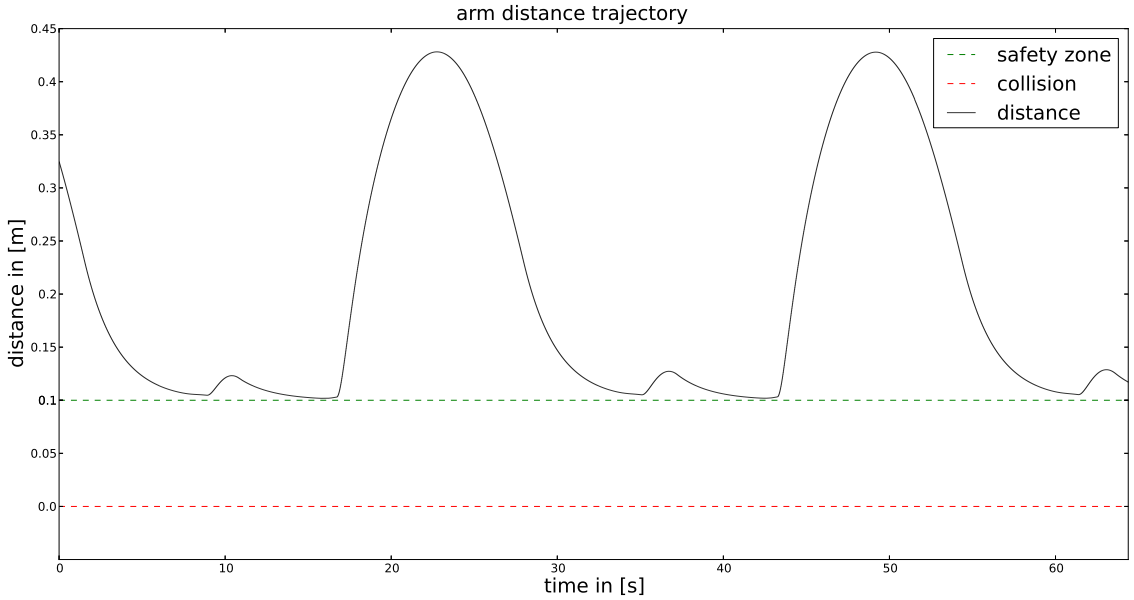


Figure 6.7.: Distance plot over time. The distance stays constantly over the safety zone, which is specified as $0.1m$. The red line indicates a distance of 0, which is obviously the result of a full contact collision.

The displacement of the actual wrist position, compared to the desired position increases, when the distance converges towards the lower boundary. Furthermore, we can clearly see

the attempt of the numerical solver to minimize the error to its best using non-constrained DOF. We can see this behavior around time $t = 10s$. Since the desired trajectory is a sine wave in Y, the direction along this axis gets constrained, when the wrist moves closer to the obstacle. It can be seen, that the two DOF, such as X and Z are not constrained and can be used to minimize the error. We can see this in the plot, whenever there is an error value in X and Z.

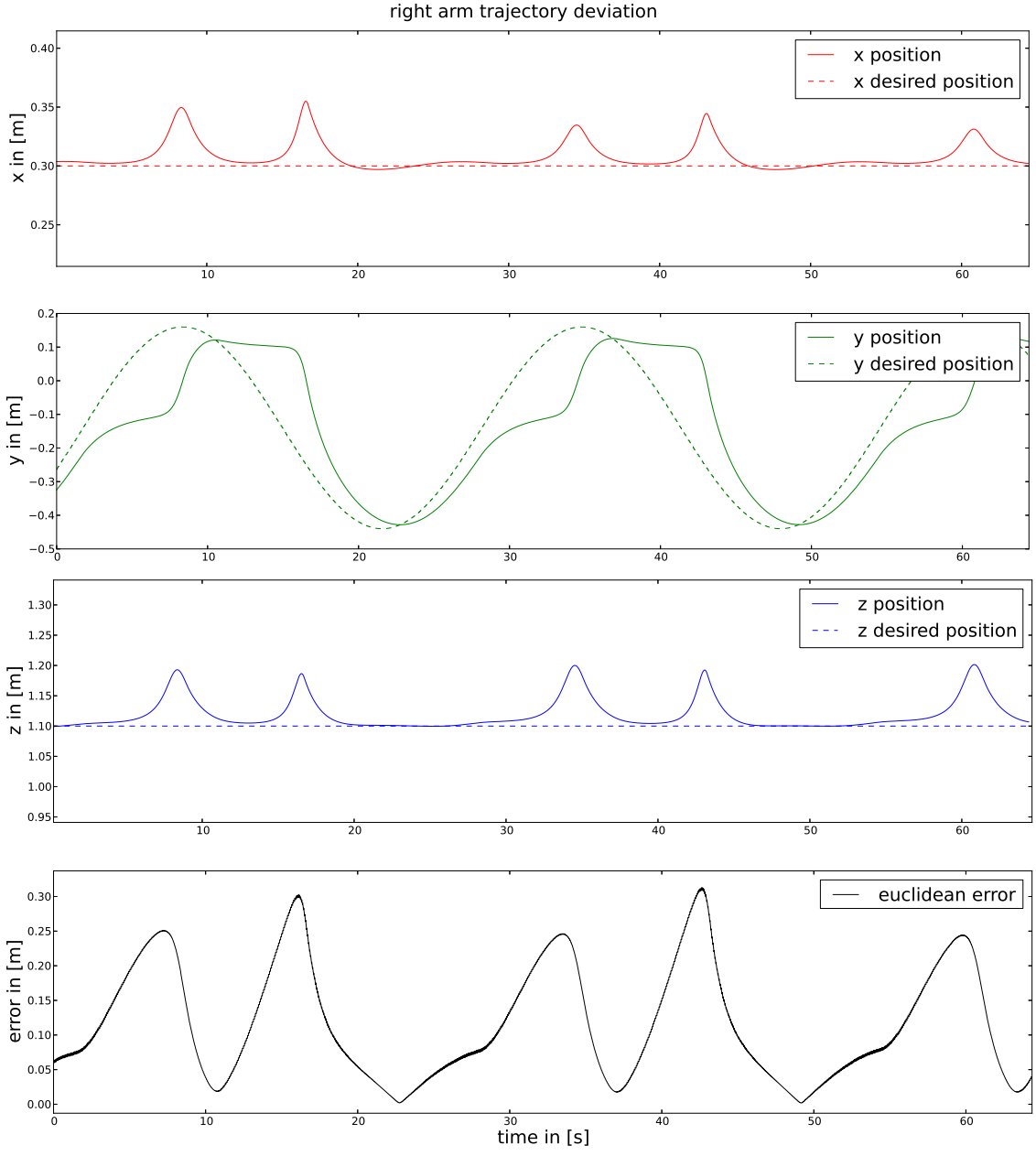
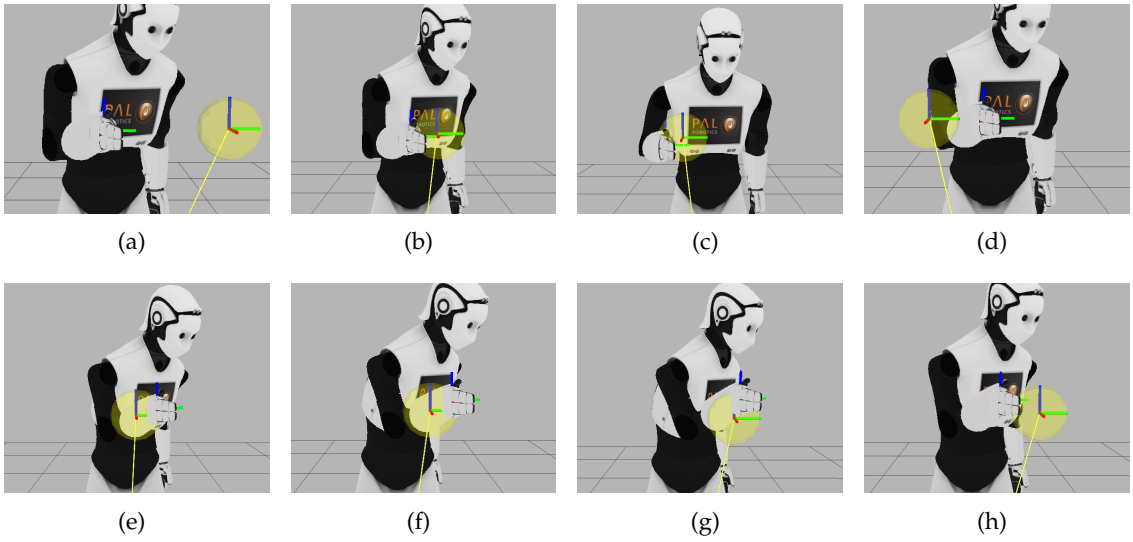


Figure 6.8.: Position plot over time. Once a collision boundary is reached, the error increases as long as the obstacle constraint. Once the obstacle got avoided, the errors minimizes again as the goal position can be reached.

Moving Object vs. Static Arm

In the previous experiment, we moved the arm along a predefined trajectory, whereas the object was at a fixed position. In this experiment, we analyze the contrary setup. In the following, we keep the wrist at a static position and move the obstacle along a path, which would finally lead to a collision. We configured the velocity damping with a $\epsilon = 10$. Equal to the previous test, k was set to 10 for the positioning task. Again, the executed trajectory is illustrated in the following sequence of graphics (figure 6.9).



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 6.9.: Sequence of screen shots for the predefined trajectory. The yellow ball represents the moving external obstacle, which is not allowed to collide with the arm. Figure a)-d) the obstacle is approaching the fixed arm from the right. When the collision boundary is reached, the arm gets pushed away from the obstacle. In order to minimize the emerging position error, the arm moves along the negative X and Z axis to avoid the obstacle. This is shown in c). Figure e)-h): The obstacle moves closes from the left side. The solution for avoiding the obstacle is a displacement along a positive Z axis.

Equally, as in the previous experiment, the collision avoidance possesses a higher priority compared to the positioning task of the right wrist, which has the lowest priority. Due to this alignment of the stack, the right wrist leaves its desired position, when the collision avoidance becomes active in figure 6.9(b). The obstacle pushes the arm away from its desired position in 6.9(c). However, the positioning task is still active, which minimizes the emerged error. The resulting minimization of the error leads into a displacement along the negative X and Z axis. Finally, the arm avoids the moving obstacle and resolves to its original desired position (figure 6.9(d)).

The second row (figure 6.9(e) - 6.9(h)) describes the counter movement, where the obstacles approaches the arm from the left side. To avoid the obstacle and to minimize the emerging error, the arm moves along a positive Z axis. We can see the exact behavior of this description in diagram 6.10.

The distance boundary d_s was again set to $0.1m$. When we have a closer look at the distance development, we expect a similar behavior as in the previous test. The minimum boundary of 0.1 is not allowed to get violated. Figure 6.11 provides information about the numerical exact distance.

Contrary to our expectations, we see that the minimal boundary is violated. The minimal distance is not $0.1m$ but $0.872m$, which is a violation of approximately 13%. Furthermore, we can see a rather high frequent oscillation, once we hit the violation.

In order to compensate for the violation as well as the oscillation, we retry the same experiments with a variation of the control gain ϵ. Firstly, we set ϵ to a really high number such as 10000 to see a tremendous chance, which can give possible insights about the reason for the violation. When we zoom in, we see the following result as depicted in figure 6.12.

The figure in 6.12 evinces a lower distance violation of $7\%, (0.937m)$ compared to the $13\%(0.872m)$ before. However, a violation still occurs with a heavily amplified oscillation, which contains huge discontinuities. These discontinuities in distance yield to a huge velocity, hence to a clearly noticeable shaking of the robot arm.

The second try to optimize this undesired behavior is to lower the gain. Thus, we redo the experiment with an ϵ of 0.01. Again, the distance development over time is plotted below (see figure 6.13).

Figure 6.13 reveals the bottleneck of this method for avoiding obstacles. The control gain ϵ is set to 0.01, which yields in no obstacle avoiding, since the minimal distance shrinks to $0.0m$, which is a full collision. Moreover, we can see that the right arm has a slow velocity, which yields, contrary to a control gain of $\epsilon = 10000$, into no heavy shaking of the robot. However, the velocity is way too small to avoid the obstacle in time.

Thus looking at the two variations, it becomes clearly that for this use case of accelerating a body part with zero initial velocity, special care has to be taken in gain tuning. When we want to provide service for any external velocities, the control gains have to be tuned according to the estimated velocity of the external obstacle. As we could see within this outcome, we have to find a trade-off between highly reactive body parts and lowest distance boundary violations.

The important issue however, is that in all variations in ϵ the boundary gets violated. We discuss the nature of this in the next section 6.3, where we analyze the reason this for in more detail.

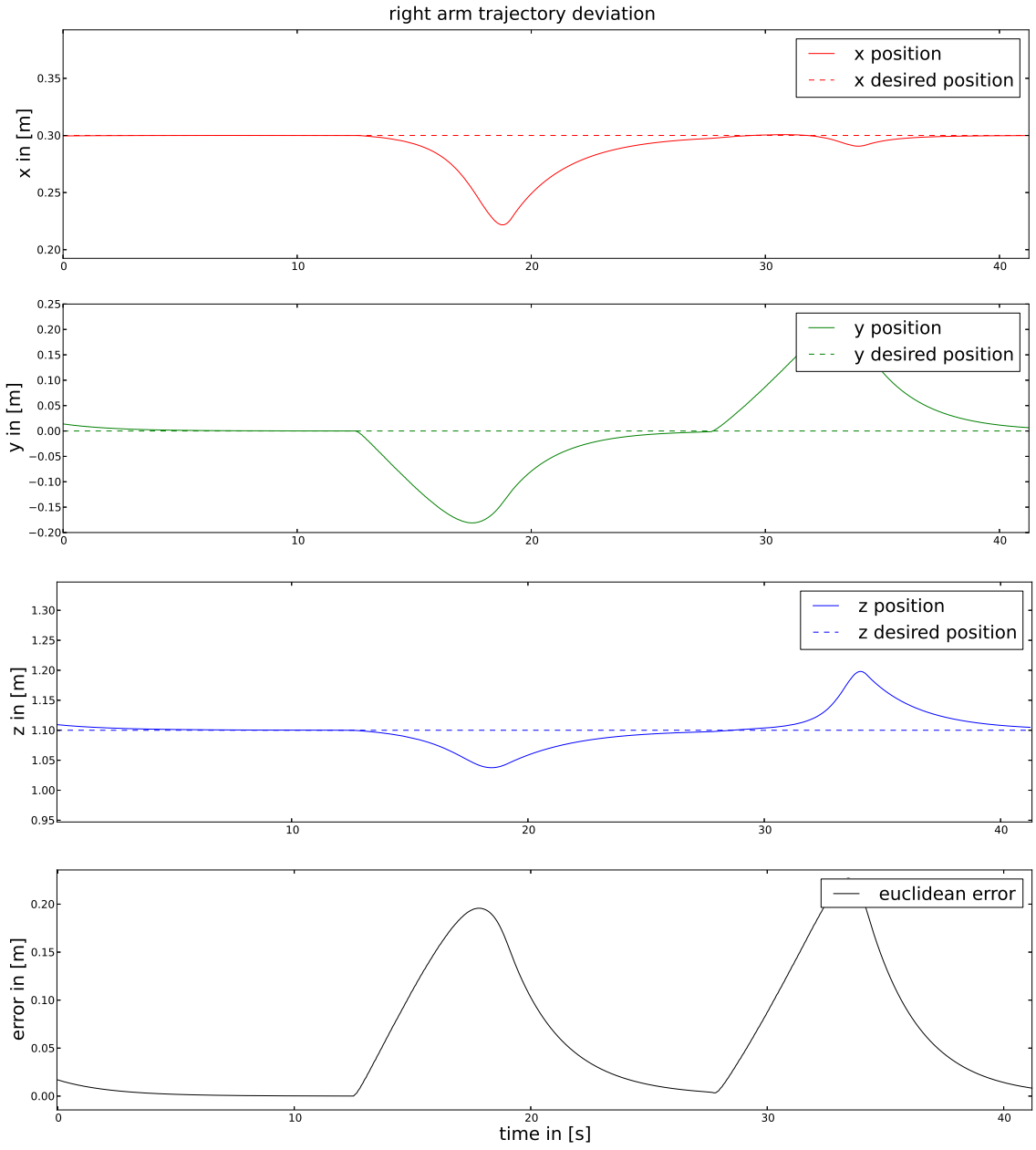


Figure 6.10.: Position plot over time. The obstacle gets avoided firstly by a negative movement in X as well as Z. On the second approach, the avoidance happens by a movement in Z.

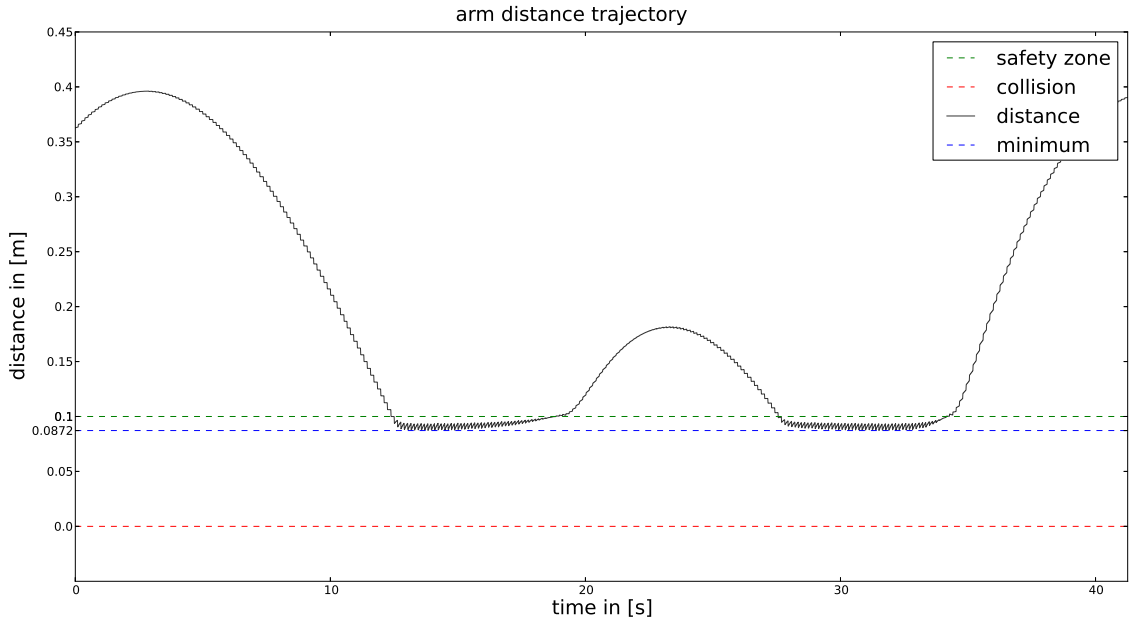


Figure 6.11.: Distance plot over time. The control gain are set to $\epsilon = 10$, as well as $k = 10$ for the positioning task. We can clearly see, that the safety distance is violated.

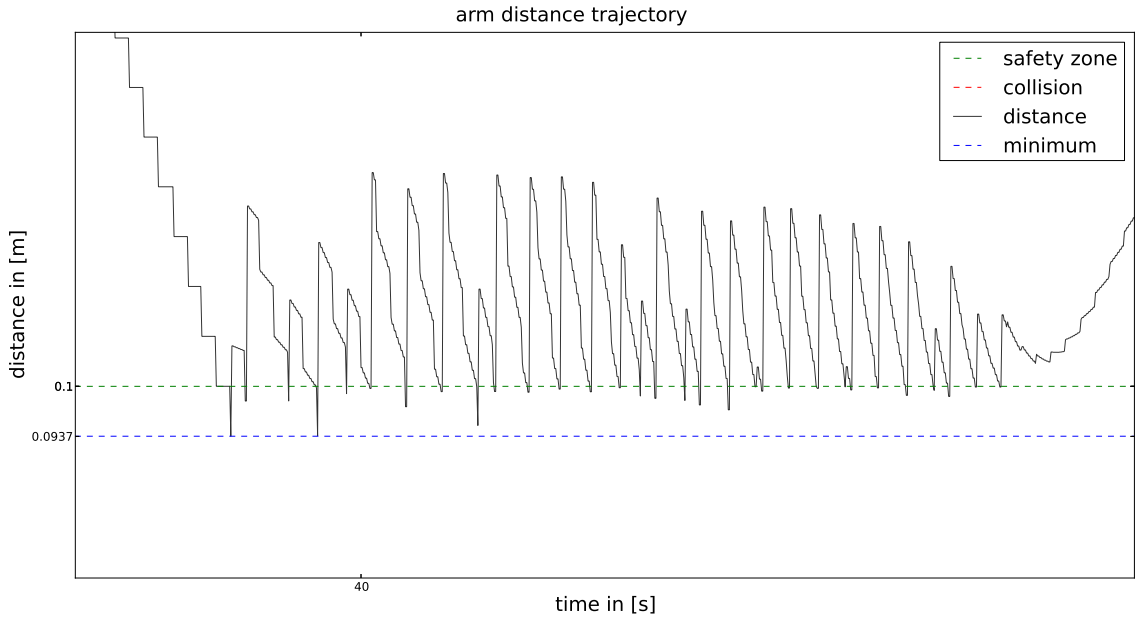


Figure 6.12.: Behavior of a huge control gain for the velocity damping task. Here, ϵ is set to 10000. We can see that the violation still occurs, but in a smaller quantity. However, the resulting oscillation increases dramatically, which results in heavy discontinuities in the velocity domain.

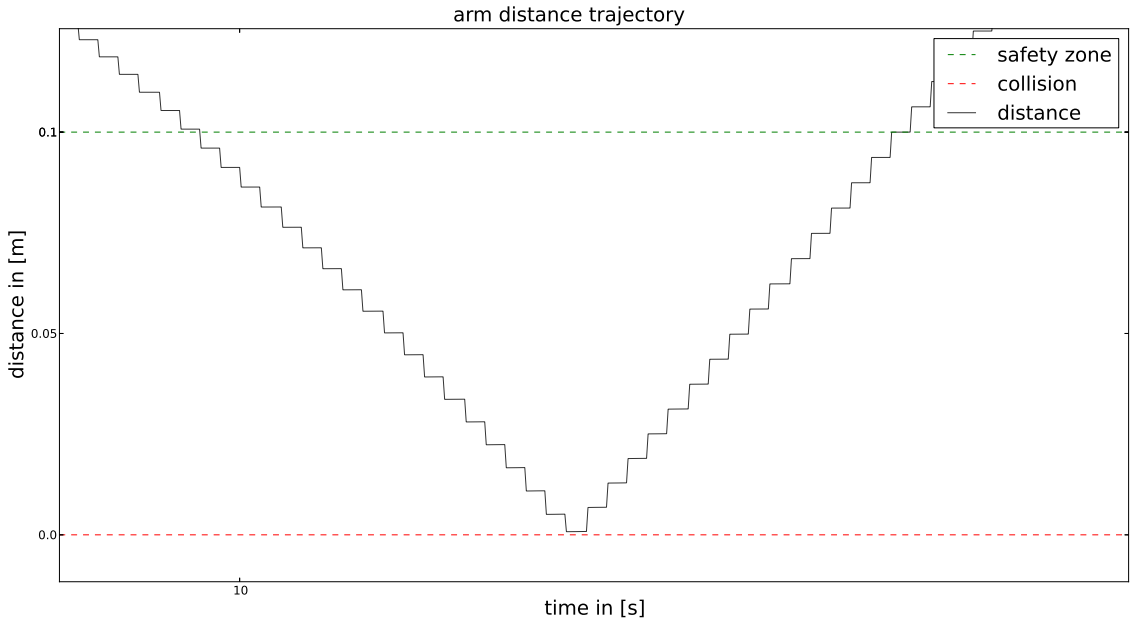


Figure 6.13.: Behavior of a low control gain for the velocity damping task. Here, ϵ is set to 0.01. We encounter a full collision and absolutely no attempt to avoid the obstacle. However, no discontinuities occur.

6.2.3. Self-Collision Avoidance

As an enhancement of the collision avoidance task for external object, we examine now an avoidance of the robots body part itself. For the following concrete tests, we refer again to the capsule decomposition provided in table 6.1. We consecutively test each group against each other.

We configure the velocity damping task according to each closest point pair, we take into consideration. Looking at figure 6.14, we illustrate the full collision matrix of possible colliding body parts with their respective closest points. In the remainder, we will test every point pair of this collision matrix. With this in mind, we test this collision matrix in a asynchronous, one-directional way. In particular, this means that, to begin with, we test the arms against the upper body as well as the head. As in the external collision avoidance, the arm tries to avoid the obstacle to its best. One-directional here denotes, that the arm avoids the upper body. However, the upper body does not avoid the arm.
Reasons for this decision were mainly to keep the number of constraints to a minimal configuration. Equally, the head and both torso links hold only one DOF, which brings just a little effect on avoiding 6 DOF of the arm. Furthermore, actively constraining the torso might have an bequeathing effect on the underlying wrist positioning task, since these links are commonly shared between both kinematic chains.

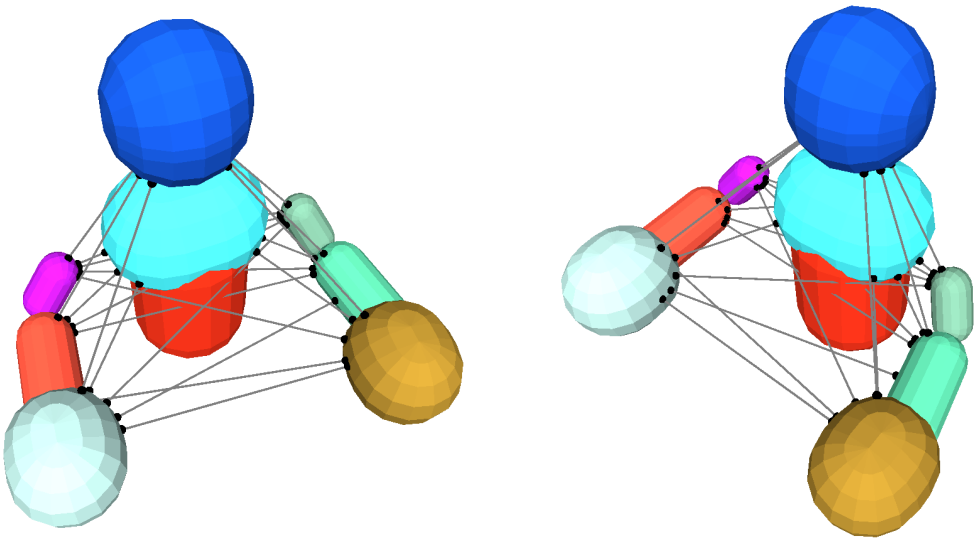
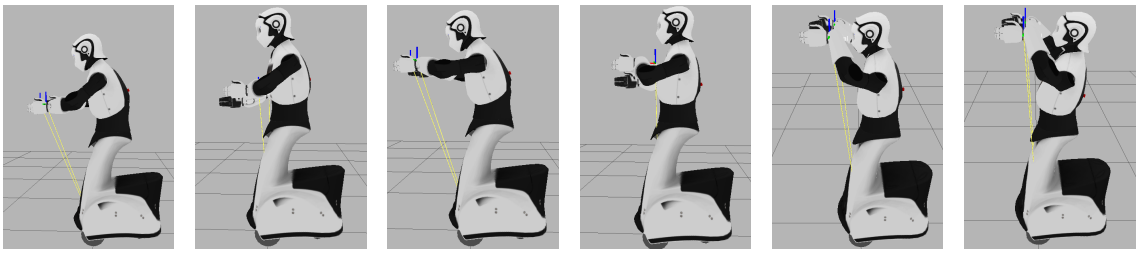


Figure 6.14.: Visualization of the full collision matrix. Every collision pair is illustrated with its respective closest point pair onto the capsules surface. In this section, we examine all possible collision pairs with each other.



(a) (b) (c) (d) (e) (f)

Figure 6.15.: Executed trajectory for the following self-collision experiment. We lead the arms into three possible collisions. Figure a)-b) would result in a collision with the lower torso, figure c)-d) equally with the upper torso. Finally, figure e)-f) illustrate a collision with the head. Thus, we describe a trajectory which alternates as a sine waves along X, Y as well as a three step increase along Z.

In the remainder of this section, a similar trajectory as the one depicted in figure 6.15 is executed. We can see, that we firstly set a goal position at the internal of the lower torso (figure a)-b)). We increase the height and repeat the same movement to provoke a collision with the upper torso (figure c)-d)). Finally, we repeat it equally with the head position (figure e)-f)).

We can see the resulting positioning diagram in figure 6.16. At approx. $t = 25s$, we shift the collision center from the lower torso to the upper torso. Similarly, we aim for colliding with the head at $t = 70s$. We do not specify this position plot in more detail in any further

experiment. We do this for simplicity reasons as the specified trajectory is mirrored for the left and right arm, which in the end results in a similar plot regardless the configuration of the positioning tasks.

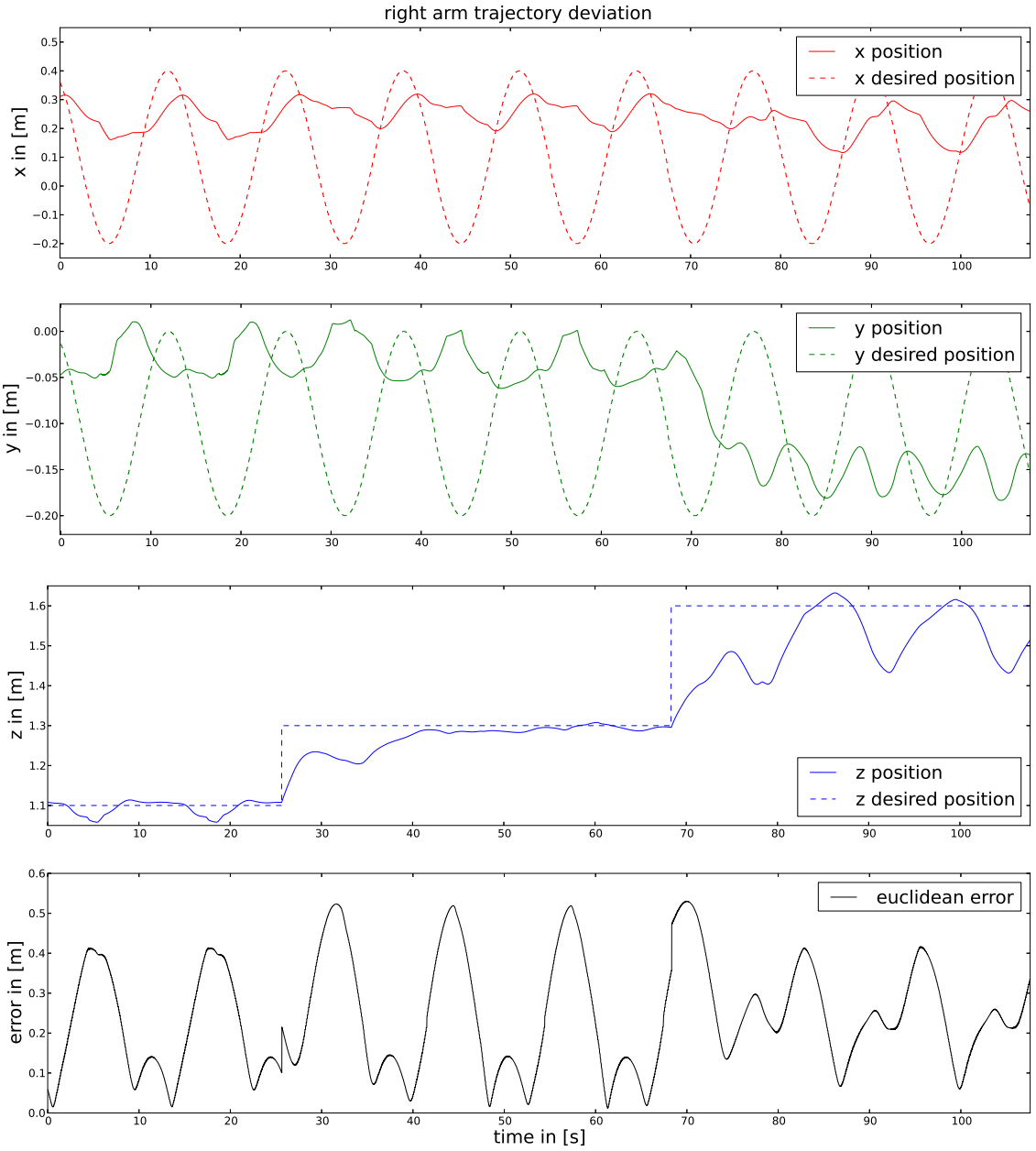


Figure 6.16.: Positioning diagram of the applied trajectory for self-collision test. As we can see in the development of along the Z axis, we increase the height in three steps: Lower torso, upper torso and head. We can assume an almost similar plot, independent whether a single arm or both are on the stack.

Arms vs. Upper Body & Head

To begin with, we test the collision between one arm and the torso, which is quite likely to occur during common movements. Depending on the desired goal position, the minimal cost trajectory often leads through the torso. Thus, we start our examination on self-collision avoidance by repeatedly specifying a goal position for the arm, which is inside the upper body and head, respectively.

The control gain for the two tasks of main interest are set to $\epsilon = 10$ for the velocity damping as well as $k = 10$ for the positioning task. A cumulated distance plot is displayed in 6.17. We see that the violation zone gets not violated except from numerical errors. We can take the violations as numerical errors since the violation does not depend on ϵ. We reproduced the experiment with different variations of ϵ with equally important results. Furthermore, looking at the numerical value of these violations, we can see a violation of $0.003m$, which essentially denotes a violation of $3mm$.

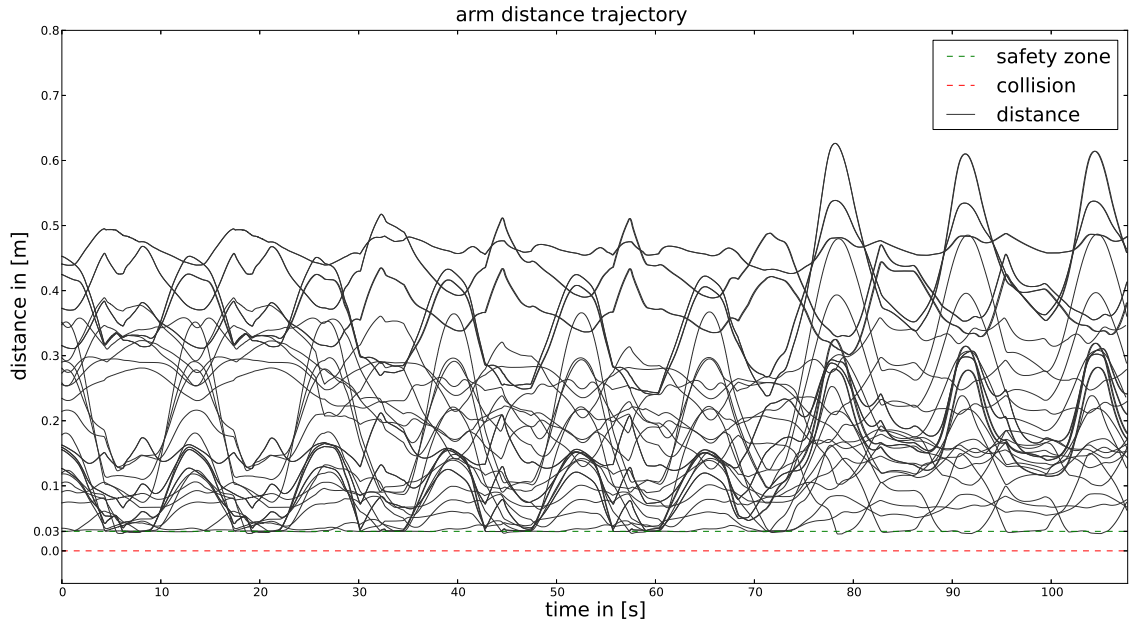


Figure 6.17.: Distance plot over all tested collisions. We can see that except numerical issues, we do not have a tremendous violation of the safety zone. We can treat those violations are numerical instabilities as the violation does not depend on any variation of ϵ. Furthermore, the maximum violation amounts to $0.003m$, which means $3mm$.

Full Collision Matrix

We introduced the self-collision trajectories to be looped in a sine wave with an three-fold increase in Z. As seen before, this yields a satisfying result, with no noticeable violations. This approach of carefully designed goal positions, with all joints in a rather safe configuration, might serve as good-to-go for most of the robotic applications.

However, we have to review the fact, that the SoT is effectively a numerical solver. That being said, we have to remark, that no offline trajectory planning is done. This fact might yield in unknown or undesired positions and joint configurations. Therefore, we redo the above experiment with a full-featured collision matrix. This means, we configure the velocity damping with all items stated in table 6.1. Hereby, we also introduce the collision between the two arms.

Furthermore, in order to expand the working area and find unplanned joint configurations, we change the applied trajectory for the left and right arm by sine waves to be randomly changed in center position and the respective 3-dimensional amplitudes. Additionally, we increased the speed of the sine wave in order to move the robot with a realistic outcome speed. The result of the distance for this experiment is shown in figure 6.18. We can easily see, that the performance is equally satisfying as with in controlled environments. Equally, we can endorse the statement of numerical violations, since the noticeable small violations still occur, yet in the same quantitative magnitude.

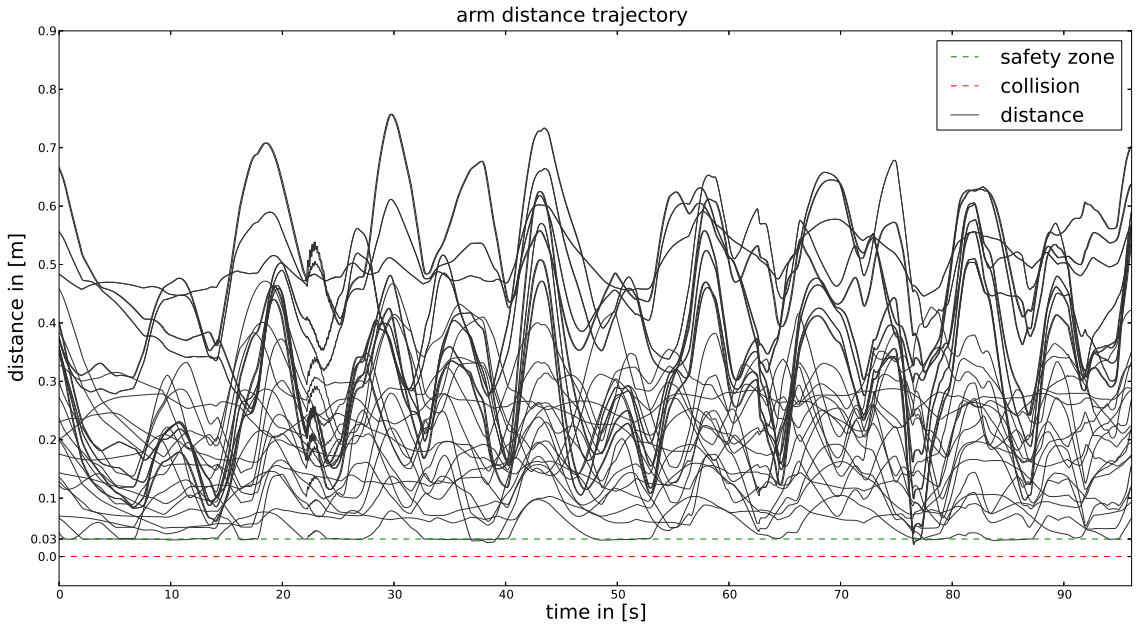


Figure 6.18.: Distance plot of randomly changed trajectory of both arms. The collision avoidance takes all possible collision pairs into consideration. The result is equally satisfying as with predefined controlled movements.

Left Arm vs. Right Arm

The result in the previous example, gave already a convincing outcome. We could assure, that all collisions (also in unexpected movements) are successfully avoided. Yet, we are mainly interested to position the hands in the most accurate way, meaning as close as possible to the desired position. Figure 6.19 extracts exclusively the resulting euclidean error of the left and right arm. As we can easily see, the desired goal position is roughly getting reached. Obviously, the reason for this is that a collision occurs before reaching the desired goal. Thus, the hierarchy works as expected as the collision avoidance takes precedence over positioning one of the arms.

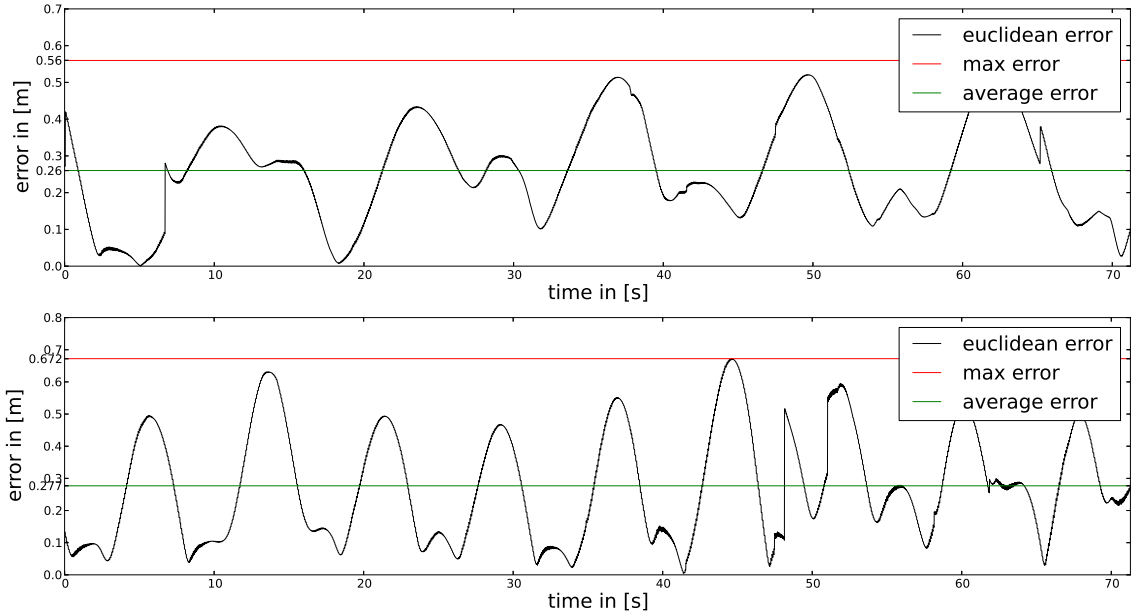


Figure 6.19.: Extraction of the emerging euclidean error. The average error is visualized in green, the maximum error value in red. We can see that both arms have a remarkable non-zero average error.

Let us have a closer look at the emerging error. The key values, such as average and maximum error value are comparably listed in table 6.3. Considering those values in the

Euclidean Error [in m]		
side	**average**	**maximum**
left	0.260	0.560
right	0.277	0.672

Table 6.3.: error value extraction of plot displayed in figure 6.19

above table raises immediately the question about the hierarchy of the two low prioritized

positioning tasks. For the above experiment, the left arm was placed over the right arm. Thus, the printed values are going along with this, yet probably not as meaningful as hoped.

To examine the priority conditions inside the hierarchy, we further test exclusively the avoidance between the two arms against each other. To do so, we define a counter directional trajectory in Y for the two arms. We examine this particular movement twice, changing the priority of the left and right arm, respectively.

Prioritizing the right arm over the left arm, yields to the resulting movement indicated in figure 6.20. We can see that according to the hierarchy level, the right[2] arm strictly encounters a smaller error compared to the left arm.

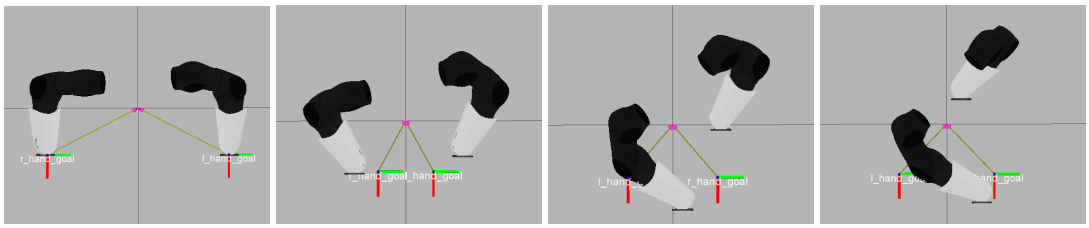


Figure 6.20.: Collision avoidance between both arms. The right arm takes precedence over the left arm.

Equally, we examined the inverse setup, having the left arm taking a higher priority than the right arm. Also, we display the sequence of movement in figure 6.21. As expected, the solution is exactly mirrored. The left arm describes a smaller error due to the higher priority.

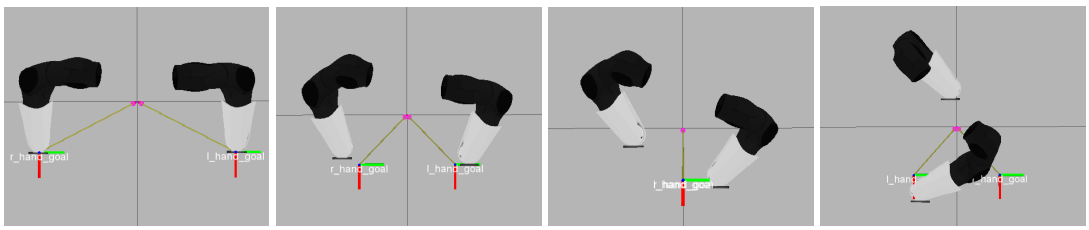


Figure 6.21.: Collision avoidance between both arms. The left arm takes precedence over the right arm.

The actual remarkable outcome of this experiment is the fact that the higher priority task encounters a deviation of its desired position, although having precedence over the respective other one. In the latter, we will suggest to name this behavior *Inversion of Hierarchy*. The synchronous collision avoidance influences the underlying tasks in a counter intuitive way, hence the name. As we can see in both displayed movements, the lower prioritized

[2]be aware that right and left are seen from the robots position. Thus, the sequence of pictures is flipped.

arm tries to avoid the higher prioritized arm to its best solution. At the same time though, the proximity between both arm rises until the lower priority encounters a clamping situation with zero velocity. This situation is equal to the distance violation we saw during the external collision avoidance tests. On the same hand, the higher prioritized arm still has to avoid every self-collision. For this reason, it also has to leave its desired position to keep the required distance threshold. Again, we refer to the next chapter 6.3, where we verbosely analyze this outcome and provide mathematical reasoning.

6.3. Results & Discussion

The overall outcome of the conducted experiments above is satisfying. We can see that, the applied method of decomposing the robots collision model into capsules yields to a smooth trajectory. Since we run those algorithms successful on the robot with $100Hz$, we can state, that the implementation is reasonably efficient. Moreover, we do not encounter any shaking of the robot body parts, even with a decent execution speed. With respect to the examined self-collision tests, we can also see a successful coverage of the complete body. Based on the placed capsules, the applied safety zone provides enough margin to bridge the transition between two adjacent capsules.

For the collision avoidance, we firstly tested our introduced method against external collision objects. External objects get strictly avoided with a continuous and optimal displacement of the actuated body part as long as this has an initial and non-zero velocity. In case of an static actuated body part with zero velocity, we have to find a reasonable trade-off between oscillating and allowed violation of the specified safety zone.

The self-collision avoidance represents the same task definition as for avoiding external objects, yet in a larger scale. For this, we configure the velocity damping task to avoid any collision with other body parts instead of one external object. This changes the task definition not to define one constraint inside the hierarchical solver but $\frac{1}{2}n^2$, where n denotes the number of entries inside the collision matrix for all capsules. Since we run the self-collision avoidance in a one-directional manner, we only consider the upper triangle of this collision matrix, hence the division by 2.

We state that all self-collisions can be avoided with great success. We encountered, besides a neglect able numerical error, that a collision avoidance based on closest point pairs of capsules yields to no self-collision. Equally important, we could execute the full collision matrix and additional lower prioritized tasks for positioning the end-effectors of both arms still in a decent speed. This generally justifies the application of this method to be sufficient enough to avoid (self-)collisions. On the same hand, it provides enough DOF to operate both arms in a reasonable speed and workspace.

6.3.1. Violations of Safety Distance

The above presented experiments show great success and give positive results for preventing self-collisions. Yet, we encountered that in certain cases, the safety zone gets violated. This misbehavior has the most impact, when the actuated body part is on a fixed position. In case of a moving collision object, the robot does not possess enough initial velocity to avoid the object in time. We similarly run into the same behavior for the self-collision avoidance, whenever clamping situations occur, which forces certain body parts to slow done until almost no velocity.

The reason for this violation comes from the fact, that the applied method for avoiding collisions does not generate any velocity. In contrast to other popular methods such as repulsive forces, we produce no force rather than restricting the existing velocity not to proceed any further in the direction of possible collisions. For convenience, we present the velocity damping formulation of 3.24 again:

$$\boldsymbol{n}^T \boldsymbol{J}(\boldsymbol{p}, \boldsymbol{q})\dot{\boldsymbol{q}} \geq -\epsilon \frac{d - d_s}{\Delta t} \tag{6.2}$$

A static position of the end-effector results in $\dot{\boldsymbol{q}} = 0$. The above formulation 6.2 thus turns into equation 6.3, which is mathematically satisfied until d_s reaches d.

$$0 \geq -\epsilon \frac{d - ds}{\Delta t} \tag{6.3}$$

In the next step, d_s becomes smaller than d, which implies an error calculation. Thus, the right hand side of the inequality in 6.4 turns into a positive expression as $d - d_s$ becomes negative. At the same hand, the directional vector n points now outside of the collision center. The overall expression flips the sign, so the inequality, which becomes finally:

$$\boldsymbol{n}^T \boldsymbol{J}(\boldsymbol{p}, \boldsymbol{q})\dot{\boldsymbol{q}} \leq \epsilon \frac{distance}{\Delta t} \tag{6.4}$$

Note, that $distance$ here implies a negative value, since $d - d_s$ denotes a negative value here. This case actively produces a positive $\dot{\boldsymbol{q}}$ and the actuated end-effector moves. However, this heavily depends on the amplifier ϵ. This being said, a velocity exists when there is an positive error between the desired and the actual position. A small ϵ just slightly amplifies the existing distance violation. Thus, a rather fast moving obstacle completely overruns the arm as shown in 6.13, because the generated velocity is simply too small. On the contrary, a heavy amplification of epsilon might catapult the actuated arm far outside of the damping constraint. This deactivated the damping inside the solver and the underlying positioning task tries to minimize the displacement error. Thus, the robot moves again towards to collision center. This finally yields to heavy oscillations between the positioning task and velocity damping.

6.3.2. Inversion of Hierarchy

Situations which occur as in the experiment of colliding two arms show that the collision avoidance has a tremendous influence on the beneath placed and thus lower prioritized tasks against each other. As we can see in figures 6.20 and 6.21, the higher prioritized arm still encounters an error in position. Concretely explained on the example, where the right arm takes precedence over the left arm: When both arms come closer, the left arm navigates towards the torso. This movement will eventually result in a clamping situation, because of the possible collisions with the right arm. Clamping situations imply again a zero velocity. We call this behavior *Inversion of Hierarchy*, since the lower priority positioning task has an remarkable impact on the above placed task.

We could see in the experiment with a full collision matrix, that those situations occur but still do not yield in a collision. To overcome this behavior, a clearly defined gain tuning has to be done for each collision pair as well as the underlying positioning tasks. At the same time, the experiments were conducted with an equally set safety zone of $0.03m$. This might lead into undesired behavior, since certain collision pairs such as the upper arm and the torso are close by construction. Therefore, we implemented the self-collision setup based on a configuration file, where for each collision pair the specific set of parameters such as ϵ and d_s are declared.

CHAPTER 7

CONCLUSION

In this Master's thesis, we successfully implemented a complete solution for collision avoidance inside a hierarchical quadratic program. We integrated the Stack-of-Tasks efficiently inside a ROS-based environment and were therefore able to run it in real time on humanoid robots, such as REEM-H and REEM-C.

We implemented a reactive collision avoidance as an inequality constraint inside the quadratic program. Hereby, the velocity along the directional vector between the closest point pair of two collision objects is damped until a specified safety distance is reached. In order to avoid any singularities in calculating this closest point pair, we carried out a complete capsule decomposition for the robots collision model. The pseudo convexity of capsules lead to a smooth and moreover continuous trajectory of point pairs. We could prove, by various experiments, that the proposed solution is sufficiently stable to avoid any self-collision as well as deliver support for external collision objects. Thus, we could with great success accomplish all three goals of this work (see chapter 1.2).

Equally, we freely admit, that the proposed method cannot be migrated as a out-of-the-box solution to different robots. Reasons for this is the strong dependency on the robot description file, which has to provide the support for capsules as a collision object. Furthermore, as we could see during the experiments, special care has to be taken for the control gain of the velocity damping. Those two configuration steps have to be calibrated for each individual robot.

The presented solution assures a safe execution of motion at all time. It ensures that all possible self-collisions are successfully avoided and the minimal distance threshold does not get violated (except numerical errors, which have to be taken into considerations). The collision avoidance thus is placed on a high level inside the hierarchy as a self-collision

should never be violated. Since the velocity damping is realized as an inequality constraint, its result denotes a solution space. The suggested method provides a solution space, which is sufficiently large to not restrict the actual manipulation goals as long as they are outside the critical safety zone. It can thus be placed transparently on the top of the stack. It prevents all self-collision, but it flexible enough to not interfere the execution of beneath placed tasks.

The implementation for the humanoid robot from PAL Robotics REEM-H comprises a basic set of activated tasks on highest level, which are strictly necessary to allow a safe usage. On highest priority, joints limits have to be in range at all time in order not to physically harm the hardware of the robot. In particular, joint limits are meant as joint position as well as joint velocity limits. This is directly followed by the self-collision avoidance tasks. With this basic stack, a safe application can be performed on lower prioritized levels.

In general, the Stack-of-Tasks achieves very convincing performance in terms of whole body motion control. The execution speed of the inverse kinematic solver is by far ahead of any current implementations. This allows a smooth and application-driven usage of humanoid robots in a whole body control. The dynamic graph enables an easy *plug-and-go* of input and output signals, which allows a straight-forward instantiation of fairly complex applications.
However, limitations arise on practical application aspects. Since the HCOD is a numerical solver, no look-ahead planning is done. This means, that even goal positions, which are clearly out of reach are solved to the best, minimizing the error between actual and goal position. This might lead to inappropriate behavior. Similarly, the DG operates in a constant control loop. Thus, emergency handling such as an immediate stop of the controller has to be handled with respect of this loop.

CHAPTER 8

OUTLOOK

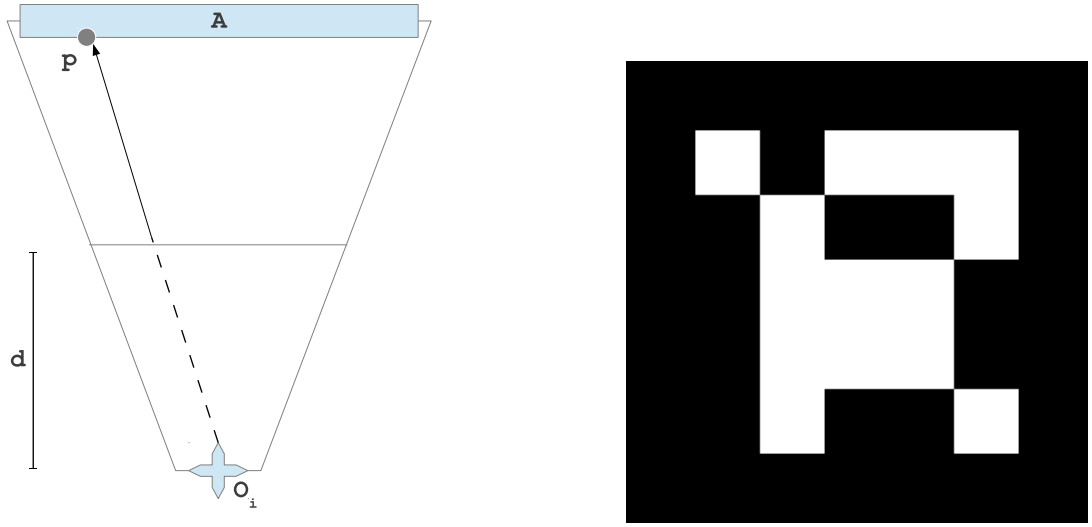
The presented method for self-collision avoidance is a passive task, which has only constraining purpose. It is not actively actuating any joints to achieve its goal. It is placed on top of the stack in order to monitor the movement of joints and prevent any collisions.

However, the real impact of the SoT comes visible, when the stack is filled with inequality and equality constraints, and the robot is controlled with the whole body. Thus, in the following chapter, we briefly introduce applications, which we execute on REEM-H. With this, we want to demonstrate the capabilities of the SoT as well as provide an outlook of what is possible with current state of the art humanoid robots.

Object Tracking

REEM-H possesses a stereo vision system as well as provides support with a Microsoft Kinect RDG-B sensor. With these vision systems we can easily implement an object tracking task. For this, we instantiate an entity, which captures the images from the vision system and provides the position of the object as an output signal.
Similarly, we instantiate a gaze task. A gaze task comprises a 2-dimensional plane $\boldsymbol{A}$, with a minimal distance $\boldsymbol{d}$ to this plane. The task is to keep the orientation of the attached joint $\boldsymbol{O}_i$ towards a point $\boldsymbol{p}$ on $\boldsymbol{A}$.

Our implementation utilizes already existing components, which are successfully implemented on REEM-H. Related work regarding object recognition was done in [36]. We performed an object recognition with the help of Aruco markers (see figure 8.1), which allows an easy positioning of objects, where this marker is attached to.



(a) Gaze Task Illustration (b) Aruco Marker

Figure 8.1.: a) Illustration of a gaze task. The 2-dimensional plane $\boldsymbol{A}$, minimal distance $\boldsymbol{d}$ and a desired point $\boldsymbol{p}$. b) Example Arcuo marker, which is used to extract a full pose or position for various objects.

A gaze task can be instantiated as a generic inverse kinematic task, which explicitly requires a Jacobian and a desired goal position for calculating the according error function. Since the vision system is attached at the head link, we plug the Jacobian of the head into the gaze task. The desired value `P` corresponds to the position extraction of the Aruco marker.

Object Grasping

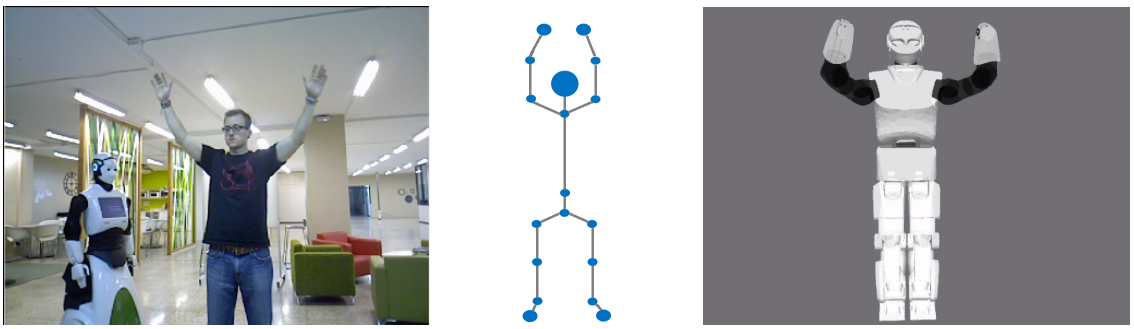
We previously developed a task for tracking an object through a vision system. The only actuated joints hereby were the ones along the kinematic chain until the head. In order to achieve object grasping, we basically instantiate a positioning task for the end-effector of one arm. Similarly, to the gaze task, we plug the position of the Aruco marker as the desired goal position for this end-effector task.

However, as in each grasping task, the main difficulty is the way the end-effector approaches the object. If we simply plug the marker position as the desired goal, we might hit the object with the manipulator, while reaching the position. We can overcome this issue by instantiating another gaze task. Contrary to the gaze task of the vision system, the place the gaze in the hand and have it orientate towards the marker. This forces the manipulator to approach the object with a correct grasping orientation.

This setup of tasks can be already seen as whole body motion control, since multiple tasks partially share the same joints. Whenever this is the case, hierarchy race conditions occur. It might be dependent on the application, which task has more priority. In our implementation, we foster the head gaze to take precedence over the grasping task, since we want to ensure to have the object in the field of view.

Motion Retargeting

As second application, we realized motion retargeting inside the SoT. Again, we could reuse existing solutions developed inside PAL [32]. Motion retargeting is mainly used to realize teleoperations. Hereby, any RGB-D sensor, such as the Microsoft Kinect, captures a 3-dimensional image of a human and extracts the skeleton image. The idea of motion retargeting is then to translate the extracted joint positions of the human into joint positions of the according robot. Figure 8.2 illustrates the process.



(a) Human movement (b) Extracted skeleton (c) Translated movement on the robot

Figure 8.2.: a) Image of the human movement b) Sketch of the extracted skeleton image c) Translated movement of the human into the robot.

We can flexibly realize teleoperations inside the SoT. The skeleton algorithms provides us with cartesian positions for each joint of the robot. Depending on how accurate we want the robot to imitate the movement of the human, we initiate a positioning task for each joint to track. Most often however, it is sufficient to track only the wrist and elbow positions, as the resulting movement of the remaining joints will be resolved quite similarly.

Dynamics

Throughout this work, we completely worked inside a kinematic domain. This means, we resolved every task with a simple kinematic model in the form of

$$\dot{\boldsymbol{x}} = \boldsymbol{J}\dot{\boldsymbol{q}} \tag{8.1}$$

This model can be shifted to a dynamics domain, which also considers velocities, inertia and masses to solve joint velocities. [44] provides a complete solution to implement dynamics task inside the SoT. The equation in 8.1 therefore becomes the Newton-Euler-Equation [18]:

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{V}(\dot{\boldsymbol{q}}, \boldsymbol{q}) + \boldsymbol{g}(\boldsymbol{q}) = \tau \tag{8.2}$$

BIBLIOGRAPHY

[1] Honda Asimo. Asimo specification. `http://asimo.honda.com/asimo-specs/`, 2014.

[2] A. Ben-Israel and T.N.E. Greville. *Generalized Inverses: Theory and Applications*. CMS Books in Mathematics. Springer, 2003.

[3] M.Z. C. S. G. LEE. *A GEOMETRIC APPROACH IN SOLVING THE INVERSE KINMATICS OF PUMA ROBOTS*. 1983.

[4] C. Chevallereau and W. Khalil. A new method for the solution of the inverse kinematics of redundant robots. In *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, pages 37–42 vol.1, Apr 1988.

[5] S Chiaverini, B Siciliano, and O Egeland. Review of the damped least-squares inverse kinematics with experiments on an industrial robot manipulator. *IEEE Trans. on Control Systems Technology*, pages 123–134, 1994.

[6] Stefano Chiaverini, Giuseppe Oriolo, and IanD. Walker. Kinematically redundant manipulators. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 245–268. Springer Berlin Heidelberg, 2008.

[7] G. Cramer. *Introduction à l'analyse des lignes courbes algébriques*. chez les frères Cramer et C. Philibert, 1750.

[8] Darpa. Darpa' atlas robot unveiled. `http://www.darpa.mil/NewsEvents/Releases/2013/07/11.aspx`, 2013.

[9] Alexander Dietrich, Thomas Wimböck, Holger Täubig, Alin Albu-Schäffer, and Gerd

Hirzinger. Extensions to reactive self-collision avoidance for torque and position controlled humanoids. In *ICRA*, pages 3455–3462. IEEE, 2011.

[10] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[11] A. Escande, N. Mansard, and P-B. Wieber. A dedicated quadradic program for fast hierarchical-inverse-kinematic resolution. In *IEEE Int. Conf. on Robotics and Automation (ICRA'10)*, Anchorage, USA, May 2010.

[12] A. Escande, N. Mansard, and P.B. Wieber. Hierarchical quadratic programming. *International Journal of Robotics Research*, October 2012.

[13] Adrien Escande, Nicolas Mansard, and Pierre-Brice Wieber. Hierarchical quadratic programming: Fast online humanoid-robot motion generation. *International Journal of Robotics Reasearch*, 2014. in press.

[14] Adrien Escande, Sylvain Miossec, Mehdi Benallegue, and Abderrahmane Kheddar. A strictly convex hull for computing proximity distances with continuous gradients. *IEEE Transactions on Robotics*, 2013. conditionally accepted.

[15] Adrien Escande, Sylvain Miossec, and Abderrahmane Kheddar. Continuous gradient proximity distance for humanoids free-collision optimized-postures. In *Humanoids*, pages 188–195. IEEE, 2007.

[16] P.J. Esrom. Robot control (the task function approach), by c. samson, m. le borgne and b. espiau oxford university press, oxford, 1991, xvii + 364, references and index (Â£50.00). *Robotica*, 9:447–448, 12 1991.

[17] B Faverjon and P Tournassoud. A local approach for path planning of manipulators with a high number of degrees of freedom. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1152–1159, 1987.

[18] R. Featherstone and D. Orin. Robot dynamics: equations and algorithms. In *Robotics and Automation, 2000. Proceedings. ICRA 2000. IEEE International Conference on*, volume 1, pages 826–834 vol.1, 2000.

[19] Korean Advanced Institute for Science and Technology. Hubo specification. `http://ohzlab.kaist.ac.kr/p_hubo2p`, 2014.

[20] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *Robotics and Automation, IEEE Journal of*, 4(2):193–203, 1988.

[21] R. S. Hartenberg and J. Denavit. *Kinematic Synthesis of Linkages*. McGraw-Hill, New

York, 1964.

[22] Fumio Kanehiro, Florent Lamiraux, Oussama Kanoun, Eiichi Yoshida, and Jean-Paul Laumond. A local collision avoidance method for non-strictly convex polyhedra. In *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008.

[23] O. Kanoun, F. Lamiraux, and P.-B. Wieber. Kinematic control of redundant manipulators: Generalizing the task priority framework to inequality tasks. (4):785–792, 2011.

[24] Oussama Kanoun, Florent Lamiraux, Pierre-Brice Wieber, Fumio Kanehiro, Eiichi Yoshida, and Jean-Paul Laumond. Prioritizing linear equality and inequality systems: application to local motion planning for redundant robots. In *ICRA 2009 - IEEE International Conference on Robotics & Automation*, pages 2939–2944, Kobe, Japon, 2009. IEEE.

[25] Lydia Kavraki, Petr Svestka, Jean claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION*, pages 566–580, 1996.

[26] Rice University Kavraki Lab. Open motion planning library: A primer.

[27] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98, Spring 1986.

[28] Oussama Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Transactions on Robotics and Automation*, RA-3(1):43–53, February 1987.

[29] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In Jerzy Neyman, editor, *Proceedings of the 2nd Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. University of California Press, Berkeley, CA, USA, 1950.

[30] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[31] Steven M LaValle. Planning algorithms, 2004.

[32] Marcus Liebhardt. Online motion retargeting for humanoid robot teleoperation. Master's thesis, Universität Stuttgart, GER, 2011.

[33] A. Liegeois. Automatic supervisory control of the configuration and behavior of multibody mechanisms. *IEEE Trans. Systems, Man, and Cybernetics*, 7(12):842–868, 1977.

[34] Hong Liu, Xuezhi Deng, and Hongbin Zha. A planning method for safe interaction between human arms and robot manipulators. In *IROS*, pages 2724–2730. IEEE, 2005.

[35] L.Sentis and O.Khatib. A whole-body control framework for humanoid operating in human environments. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 2641–2648, Orlando, FL, USA, May 2006.

[36] Bence Magyar. Implementing visual perception tasks for the reem robot. Master's thesis, Etövös Loránd University, HU, 2012.

[37] N. Mansard, O. Khatib, and A. Kheddar. A unified approach to integrate unilateral constraints in the stack of tasks. *IEEE Transaction on Robotics*, 25(3), June 2009.

[38] N. Mansard, O., P. Evrard, and A. Kheddar. A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The stack of tasks. In *International Conference on Advanced Robotics (ICAR)*, page 119, June 2009.

[39] A. Martinez and E. Fernández. *Learning ROS for Robotics Programming*. Packt Publishing, 2013.

[40] Wim Meeussen, Adolfo Rodriguez Tsouroukdissian, and Dave Coleman. Ros-control. `http://wiki.ros.org/ros_control`, 2014.

[41] K.G. Murty. *Linear programming*. Wiley, 1983.

[42] Jorge Nocedal and Stephan J. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, New York, NY, 2. ed. edition, 2006.

[43] Jia Pan, Sachin Chitta, and Dinesh Manocha. Fcl: A general purpose library for collision and proximity queries. In *ICRA*, pages 3859–3866. IEEE, 2012.

[44] Oscar Efraín Ramos Ponce. Generation of complex dynamic motion for a humanoid robot. Master's thesis, LAAS-CNRS, FR, 2011.

[45] Lorenzo Sciavicco and Bruno Siciliano. *Modelling and Control of Robot Manipulators*. Springer, January 2000.

[46] J.M. Selig. Lie groups and lie algebras in robotics. In Jim Byrnes, editor, *Computational Noncommutative Algebra and Applications*, volume 136 of *NATO Science Series II: Mathematics, Physics and Chemistry*, pages 101–125. Springer Netherlands, 2004.

[47] Fumi Seto, Kazuhiro Kosuge, and Yasuhisa Hirata. Self-collision avoidance motion control for human robot cooperation system using robe. In *IROS*, pages 3143–3148. IEEE, 2005.

[48] Bruno Siciliano. Kinematic control of redundant robot manipulators: A tutorial. *Journal of Intelligent and Robotic Systems*, 3(3):201–212, 1990.

[49] Bruno Siciliano, Lorenzo Sciavicco, and Luigi Villani. *Robotics : modelling, planning and control*. Advanced Textbooks in Control and Signal Processing. Springer, London, 2009. 013-81159.

[50] Bruno Siciliano and J-JE Slotine. A general framework for managing multiple tasks in highly redundant robotic systems. In *Advanced Robotics, 1991.'Robots in Unstructured Environments', 91 ICAR., Fifth International Conference on*, pages 1211–1216. IEEE, 1991.

[51] Pal Robotics S.L. Reem-c specification. `http://www.pal-robotics.com/en/robots/reem-c`, 2014.

[52] O. Stasse, A. Escande, N. Mansard, S. Miossec, P. Evrard, and A. et Kheddar. Real-time (self)-collision avoidance task on a HRP-2 humanoid robot. In *icra08*, Las Passadenas, USA, May 2008.

[53] Ioan Alexandru Sucan and Lydia E. Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Algorithmic Foundation of Robotics VIII (Proceedings of Workshop on the Algorithmic Foundations of Robotics)*, volume 57, pages 449–464, Guanajuato, Mexico, 2009. STAR, STAR.

[54] Hisashi Sugiura, Michael Gienger, Herbert Janssen, and Christian Goerick. Real-time self collision avoidance for humanoids by means of nullspace criteria and task intervals. In *Humanoids*, pages 575–580. IEEE, 2006.

[55] Sebastian Thrun. Probabilistic algorithms in robotics. *AI Magazine vol*, page 2000.

[56] Robert F. Tobler and Stefan Maierhofer. A mesh data structure for rendering and subdivision.

[57] Gino Van den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2):7–25, March 1999.

[58] D. E. Whitney. The mathematics of coordinated control of prosthetic arms and manipulators. *Trans. ASME Journal of Dynamic System, Measures and Control*, 94, 1972.

[59] H P Xie, R V Patel, S Kalaycioglu, and H Asmer. Real-time collision avoidance for a redundant manipulator in an unstructured environment. In *in International Conference on Intelligent Robots and Systems*, page pages, 1998.

[60] H. Hanafusa Y. Nakamura. Optimal redundancy control of robot manipulators. *International Journal of Robotics Research*, 1986.