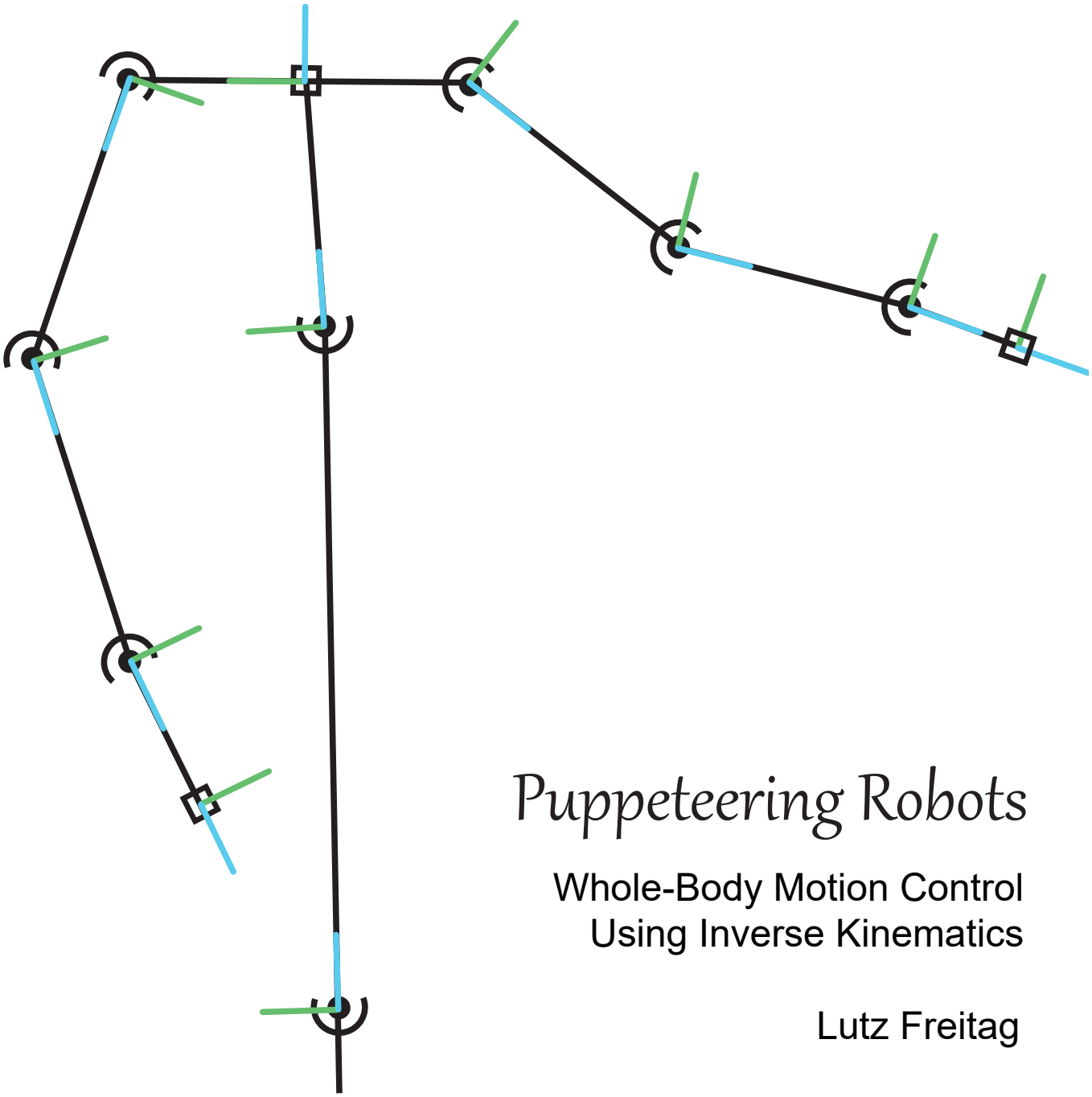
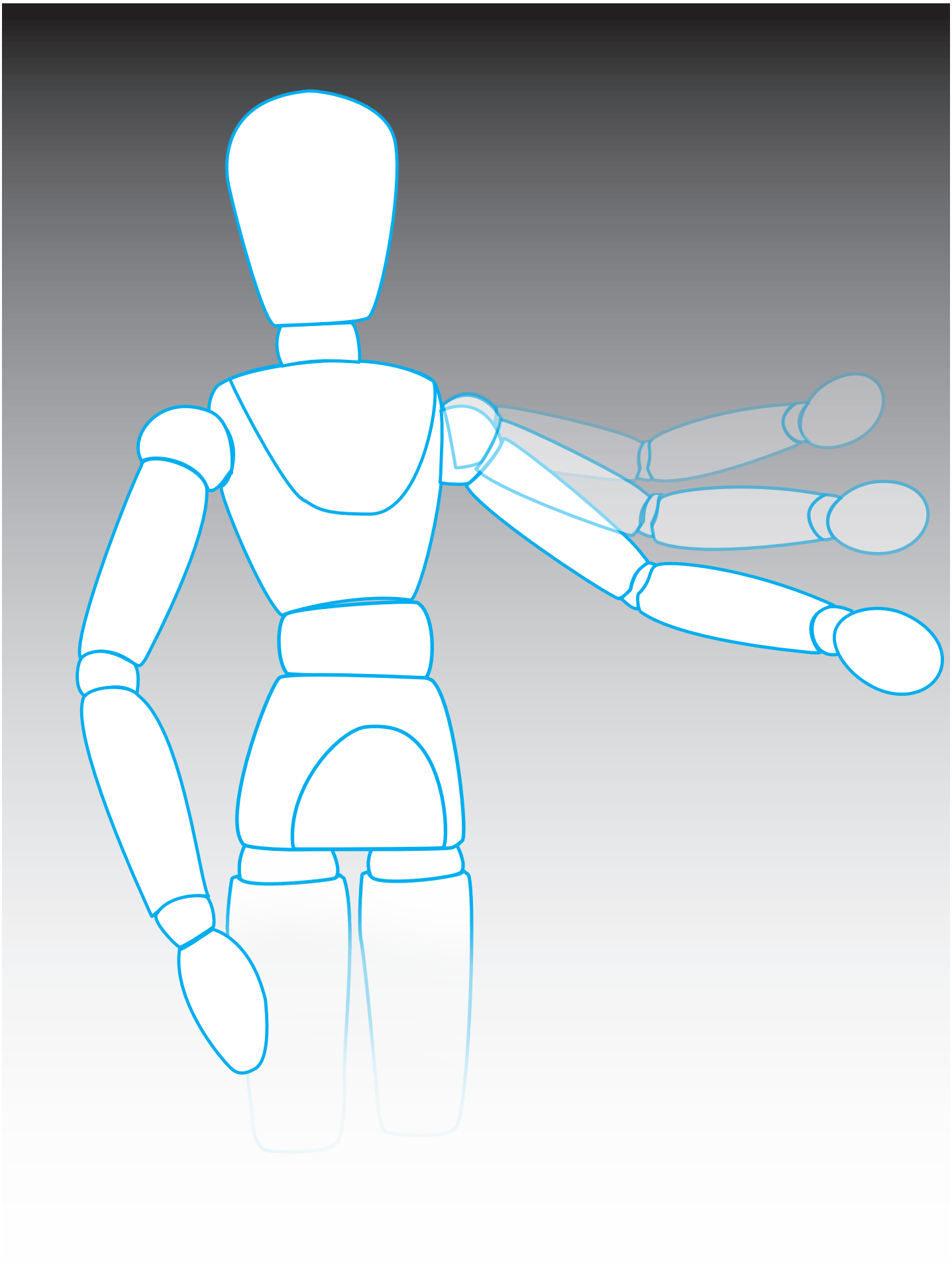


Puppeteering Robots

Whole-Body Motion Control Using Inverse Kinematics

Lutz Freitag



Puppeteering Robots

Whole-Body Motion Control Using Inverse Kinematics

Lutz Freitag
lutz.freitag@fu-berlin.de

Master's Thesis
Freie Universität Berlin
Fachbereich Mathematik und Informatik

Advisors
Prof. Dr. Raúl Rojas
Prof. Dr. Daniel Göhring

Berlin, 9th Jun, 2016

Declaration of Academic Honesty

I hereby declare that this master's thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the bibliography.

Berlin, 9th Jun, 2016

Lutz Freitag

_____

Abstract

This master's thesis provides a conclusive derivation of the dampened least-squares approach to inverse kinematics. Even though this approach is widely researched the underlying mathematics are not well covered. The dampened least-squares approach is proven to be very versatile. It gives the ability to express motion in spacial coordinates which renders motion creation much more feasible in contrast to motion generation in joint-space. Inverse kinematics can also be utilized as an abstraction to robot models and structures since motion in task space can be easily transferred to different robots. In addition to the derivation this thesis discusses extensions to the performance in terms of numerical stability, its uses for nontrivial robot structures (e.g., loopy robots) and the incorporation of nonlinear constraints. Furthermore, this thesis shows that the dampened least-squares inverse kinematics outperforms analytical approaches in terms of general usability.

Acknowledgment

I want to express my gratitude to the entire FUmanoids team and all its former members. As part of the team I had the possibility to gain knowledge in a lot of areas which I would have missed otherwise – and I had a tremendous amount of fun! Also, I want to thank Marc Toussaint for inspiring me to get into the field of inverse kinematics, Raúl Rojas for supporting the FUmanoids, Daniel Göhring for his everlasting patience when I want to discuss ideas, Simon Gene Gottlieb for all the crazy adventures we had, Daniel Seifert for so many things I cannot list here and all robots I had the chance to work with during my time with the FUmanoids.

Go FUmanoids!

Contents

List of Figures

Listings

1 - Introduction

In many modern manufacturing processes robots increasingly gain popularity. For example, in car manufacturing whole assembly lines are already completely automated by robots. Among other tasks robots weld, cut and hold workpieces in a way that generates an overall streamlined manufacturing process. What appears to be an overwhelmingly complex movement of several robots interacting is usually a set of individual movements. The set of all individual movements is orchestrated and synchronized in a way that gives the impression of robots working in conjunction. At the very basic level robot's movements are programmed in terms of joint movements. This means for each time step each joint of every robot has a predefined value.
While this expression of movement creates suitable trajectories for manufacturing processes it is not necessarily intuitive. Humans usually express movements in terms of body parts that are spatially moved. Expressing where a hand has to move to in order to grasp something is much more intuitive than expressing how the joints have to be moved to achieve the same movement. Furthermore, a system can be built that compensates for different robot structures. The mapping from a desired body part's location to a posture is called *inverse kinematics* and is quite extensively researched. However, even though inverse kinematics has been focused by many researchers there is still a lack in the mathematical foundation.
This master's thesis provides a conclusive derivation of the *dampened least-squares* inverse kinematics approach. An optimality criterion is introduced from which the basics of inverse kinematics are derived. Some – already commonly utilized – extensions are discussed and proven to be optimal with respect to the optimality criterion and some extensions are proven to be non-optimal. Additionally another extension that allows the utilization of *loopy robots* is introduced. Lastly, approaches to nonlinear constraints (e.g., *joint-limits*) are discussed and a novel technique to resolve those constraints is proposed.

2 - Forward Kinematics

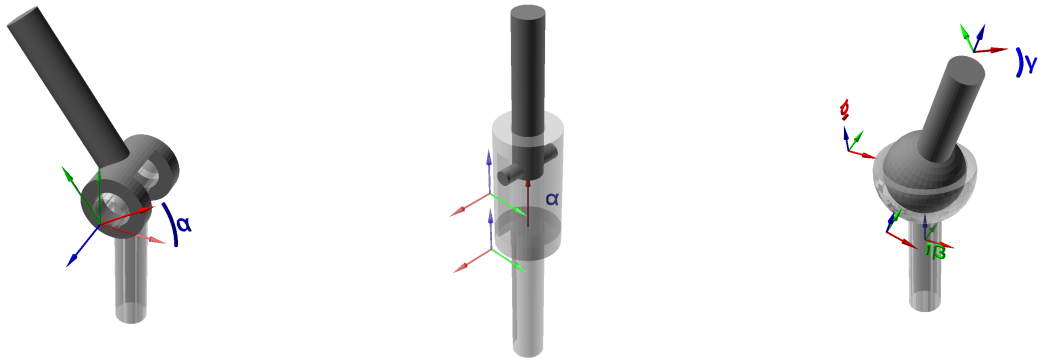
Forward kinematics is the answer to the question:

Given a robot's joint configuration:
Where is body part A located and oriented with respect to body part B?

With that in mind tools to represent perspectives as well as locations and orientations of body parts can be derived. This will give the spatial representation of postures.

2.1 Representations of Rigid Robotic Systems

Commonly robots are built in a *treelike structure* [1]. Nodes represent body parts that are themselves connected to other parts of the body creating a loop-free topology.
The most widespread representation of robotic systems used in the literature as well as in software are robots composed of rigid *links* and *joints*. Joints connect links and represent the robot's degrees of freedom. The concept of joints and links, however, originates in physics simulations where links represent single bodies (with physical properties). Joints connect the bodies thus limit their degrees of freedom in terms of spatial movements. Joints do not necessarily need to be articulated; they can represent passive connections between links as well. Also, they represent the actual type of connection between body parts e.g., prismatic, revolute, combined-revolute etc. Some examples of joint-types are displayed in figure 2.1.



(a) hinge/revolute joint (b) piston/prismatic joint (c) ball/universal joint

Figure 2.1: Examples of joint types

For each degree of freedom a joint has a *value* – the angle of a revolute joint, three angles for a ball-joint or the stroke of a piston. The set of all joint-values of a robot is the robot's *configuration* or *posture*.
The joint-link-representation would also allow the construction of *loopy* systems. Those are systems that do not have a treelike structure but contain loops in terms of graphs. When using any spanning tree from the loopy system an equivalent system can be created. Chapter 6 shows that treelike robots can be made to behave exactly the same way as loopy robots. Without any loops the robotic system has a tree structure which guarantees there exists exactly one direct path between two body parts.
To describe the spatial posture of a robot links are actually not necessary. An equivalent construction of a `joint1-link-joint2` system can be accomplished by letting `joint1` have `link` and `joint2` as children. In the latter system `link` can be expressed as a dummy-joint without any degrees of freedom. The connection between `joint1` and `link` now does not serve any articulated purpose but dummy-joints can be used to represent useful body parts like fingertips or cameras which are offset-attached to an articulated joint.
Lastly, without loss of generality, joints can be split into a series of *elementary joints* [1]. E.g., a ball joint can be constructed from three revolute joints. By splitting combined joints into elementary joints an order has to be be defined in which they act upon the affected body parts' positions and orientations. An example how a ball joint can be split into three subsequent transformations with the rotation angles α, β, γ is shown in figure 2.1c. Note that the coordinate frames are offset to the actual point of rotation to better display the transformations. Any node or joint in the previously mentioned tree structure represents a body part. From here on the term *effector* will be used for nodes.

2.2 Homogeneous Transformations

Nested coordinate systems can be interpreted as transformations that convert points from the perspective of coordinate system A to another coordinate system B.

Example

Figure 2.2 displays a 3D-coordinate system B that is located at $o_B = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$ and with $\alpha_z = 45°$ counterclockwise rotation around the z-axis as seen from a coordinate system A.
A point p_B seen from coordinate system B can be transformed into coordinate system A by applying:

$$\begin{aligned} p_A = & \underbrace{R_B * p_B}_{\text{rotate } p_B \text{ to A}} + \underbrace{o_B}_{\text{offset of } B} \\ p_A = & \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix} * p_B + \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \end{aligned} \tag{2.1}$$

R_B is the rotation matrix that rotates points from B to A and o_B is the location of the coordinate frame B as seen from A.

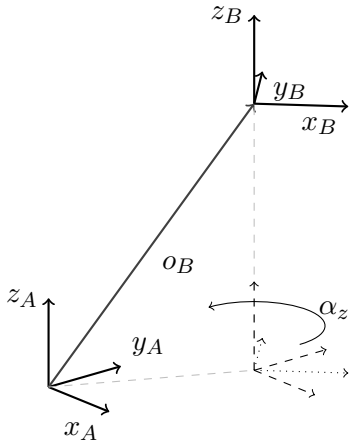


Figure 2.2: Coordinate system B defined in A

To get an equivalent but more compact formulation of equation (2.1) any point p can be augmented with an additional row with the value 1 to get $\hat{p}$ and introduce the *homogeneous Transformation* ${}^{A}T_{B}$:

$${}^{A}T_{B} = \begin{pmatrix} R_B & o_B \\ 0 & 1 \end{pmatrix} \qquad \text{and} \qquad \hat{p} = \begin{pmatrix} p \\ 1 \end{pmatrix}$$

${}^{A}T_{B}$ is a commonly used notation [2] that will also be used thoughout this thesis to denote transformations from a coordinate frame B into the coordinate A.
With that equation (2.1) can be reformulated:

$$\begin{pmatrix} p_A \\ 1 \end{pmatrix} = \hat{p}_A \quad = \begin{pmatrix} R_B & o_B \\ 0 & 1 \end{pmatrix} * \hat{p}_B = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 1 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} p_B \\ 1 \end{pmatrix} \tag{2.2}$$

Homogeneous transformations can also be utilized to express relationships (perspectives) of coordinate frames that are indirectly connected. Here transformations are multiplied in the sequence the coordinate frames are related to each other as shown in equation (2.5). Since homogeneous transformations can be used to unify a sequence of transformations into a single transformation they can also be decomposed into a subsequent series of rotations and translations.

$${}^{A}T_{B} = \begin{pmatrix} {}^{A}R_B & {}^{A}o_B \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & {}^{A}o_B \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} {}^{A}R_B & 0 \\ 0 & 1 \end{pmatrix} \tag{2.3}$$

This means every transformation of the above form can be easily deconstructed into a translation and a rotation that need to be applied in that order to create the original transformation. This also means that any complex joint type can be constructed by a set of primitive joints – revolute and prismatic joints – which are working serially.

2.3 Logic Representation of Articulated Systems

A robot's posture may be defined as a vector of joint configurations q (e.g., angles, strokes, etc.) where each element corresponds to one degree of freedom of the robot. Two transformations can be defined for each joint:

- extrinsic transformation ${T_{ext}}$
  The offset position and orientation where a joint is attached to its parent effector. This matrix is not dependent on the robot's configuration and defines the robot's effector layout (offsets). The matrix in equation (2.2) is an example of an extrinsic transformation.
- intrinsic transformation ${T_{int}}$
  The transformation that is applied by the joint itself. For revolute joints this is a rotation matrix and for prismatic joints a translation matrix. For fixed effectors this matrix is the identity.

For convenience the extrinsic transformation should be defined in a way that the intrinsic transformation becomes the identity for $q_i = 0$. That is when the value at the i-th joint is 0. This will prove to be useful in chapter 4.

$$ {}^{i}T_{j} = {}^{i}T_{j_{ext}} * T_{j_{int}}(q_j) \tag{2.4} $$

The matrix ${}^{i}T_{j}$ transforms from the j-th into the i-th coordinate system where the i-th effector is the j-th effector's parent. It is easy to see that the description is *forward* with respect to the tree structure – when traversing from effector i to j (e.g., from the robot's root to one of its children) the static offset transform (extrinsic) has to be applied first and then the intrinsic transformation. Furthermore, ${}^{i}T_{j}$ solely depends on q_j. An example for this sequence of transformations is given in figure 2.3.

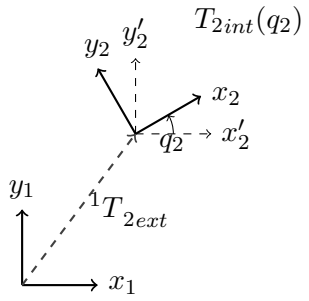


Figure 2.3: Example of an extrinsic and intrinsic transformation

Note that the inverse (*backward*) transformation is:

$$ {}^{j}T_{i} = {}^{i}T_{j}{}^{-1} = \left({}^{i}T_{j_{ext}} * T_{j_{int}}(q_j)\right)^{-1} = T_{j_{int}}(q_j)^{-1} * {}^{i}T_{j_{ext}}{}^{-1} = T_{j_{int}}(q_j)^{-1} * {}^{j}T_{i_{ext}} $$

The intrinsic transformations for the joints in figure 2.1 are:

- hinge/revolute joint rotating around unit vector $\vec{v}$ with angle α:

  $$ c = \cos\alpha $$

$s = \sin\alpha$

$$\begin{pmatrix} \vec{v}_x^2(1-c)+c & \vec{v}_x\vec{v}_y(1-c)-\vec{v}_z s & \vec{v}_x\vec{v}_z(1-c)+\vec{v}_y s & 0 \\ \vec{v}_x\vec{v}_y(1-c)+\vec{v}_z s & \vec{v}_y^2(1-c)+c & \vec{v}_y\vec{v}_z(1-c)-\vec{v}_x s & 0 \\ \vec{v}_x\vec{v}_z(1-c)-\vec{v}_y s & \vec{v}_y\vec{v}_z(1-c)+\vec{v}_x s & \vec{v}_z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- piston/prismatic joint with jerk α and direction $\vec{d}$:

$$\begin{pmatrix} 0 & 0 & 0 & \\ 0 & 0 & 0 & \alpha\vec{d} \\ 0 & 0 & 0 & \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- ball/universal joint with angles α, β and γ:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  Note that the transformations are sequential. The rotation axis of the second transformation is defined by the first rotation while the rotation axis of the third transformation is defined by the first two transformations.

To calculate the (*forward*) transformation from an arbitrary effector i to any other body part j the function ϕ is defined as:

$$^{i}\phi_j(q) = {}^{i}T_j = \prod_{\substack{k,l \\ k \text{ parent of } l}} {}^{k}T_l \tag{2.5}$$

Here one traverses along the *path* (also referred to as *kinematic chain*) from the i-th effector to the j-th effector and multiplicatively accumulate each subsequent transform. The order in which the transforms are multiplied is important and must follow the path. Since the robot is constructed as a tree structure there exists only one path for each i, j combination.

Example

Figure 2.4 displays the structure of a two dimensional robot with two arms. There are three revolving degrees of freedom at the effectors 2, 3 and 4. Effectors 3 and 4 are attached to effector 2 behind its intrinsic rotation. Effector 2 is attached to a fixed base at the origin. The forward extrinsic transformations are drawn along the topology vectors in the figure. Note that due to readability the intrinsic transformations are not displayed but depicted as rotations within each node as q_i. The transformation ${}^{3}T_4$ can be constructed as:

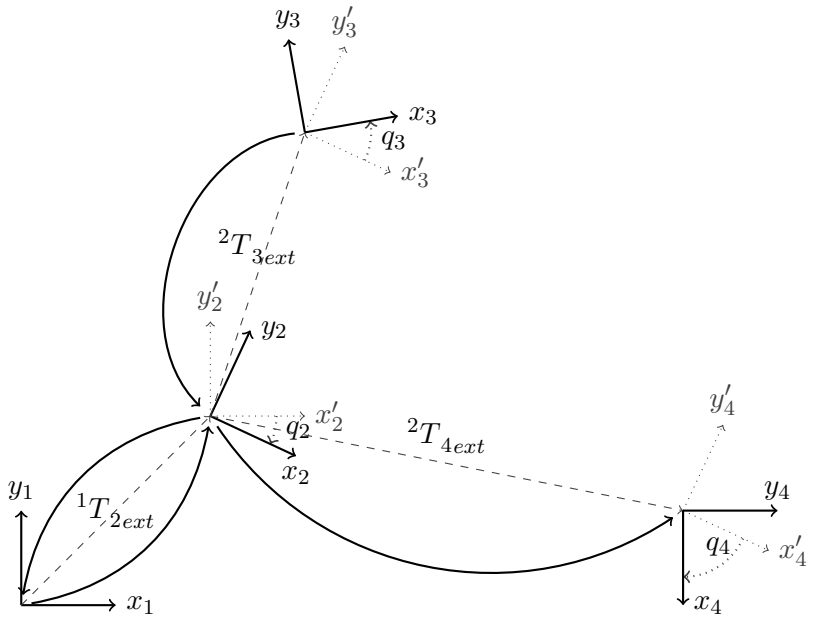


Figure 2.4: Calculation of the transformation from joint 2 to joint 4

$$
\begin{aligned}
{}^3T_4 &= {}^3T_1 && * {}^1T_4 \\
&= {}^1T_3{}^{-1} && * {}^1T_4 \\
&= \left({}^1T_2 * {}^2T_3\right)^{-1} && * {}^1T_2 * {}^2T_4 \\
&= \left({}^1T_{2ext} * T_{2int} * {}^2T_{3ext} * T_{3int}\right)^{-1} && * {}^1T_{2ext} * T_{2int} * {}^2T_{4ext} * T_{4int} \\
&= T_{3int}{}^{-1} * {}^2T_{3ext}{}^{-1} * T_{2int}{}^{-1} * {}^1T_{2ext}{}^{-1} && * {}^1T_{2ext} * T_{2int} * {}^2T_{4ext} * T_{4int} \\
&= T_{3int}{}^{-1} * {}^2T_{3ext}{}^{-1} && * {}^2T_{4ext} * T_{4int} \\
&= {}^2T_3{}^{-1} && * {}^2T_4
\end{aligned}
\tag{2.6}
$$

As shown above the traversal to the robot's root node is not necessary since those forward and backward transformations cancel each other out. The remaining transformations are the traversal along the shortest path between node 3 and 4 being the sequence $\{3 \to 2; 2 \to 4\}$. Equation (2.6) also shows that the intrinsic transformation within the node 2 does not affect the transformation from 3 to 4.
The methods described in this section allow the calculation of locations and orientations of body parts with respect to other body parts. An orientation and location is expressed with a single homogeneous matrix.

2.4 Workspaces

The space of all possible locations and orientations of an effector is called *workspace*. It is the space where an effector b can be moved to and rotated with respect to another effector a.
The matrix $J_{{}^a\phi_b(q)}$ maps the robots degrees of freedom to the degrees of freedom of ${}^a\phi_b$ when

being in posture q [3].

$$J_{{}^a\phi_b(q)} = \frac{d}{dq}{}^a\phi_b(q) = \begin{pmatrix} \frac{\delta}{\delta q_1}{}^a\phi_b(q)_1 & \frac{\delta}{\delta q_2}{}^a\phi_b(q)_1 & \cdots & \frac{\delta}{\delta q_n}{}^a\phi_b(q)_1 \\ \frac{\delta}{\delta q_1}{}^a\phi_b(q)_2 & \frac{\delta}{\delta q_2}{}^a\phi_b(q)_2 & \cdots & \frac{\delta}{\delta q_n}{}^a\phi_b(q)_2 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta}{\delta q_1}{}^a\phi_b(q)_k & \frac{\delta}{\delta q_2}{}^a\phi_b(q)_k & \cdots & \frac{\delta}{\delta q_n}{}^a\phi_b(q)_k \end{pmatrix} \tag{2.7}$$

The index i at ${}^a\phi_b(q)_i$ denotes the i-th degree of freedom of ${}^a\phi_b(q)$. The matrix $J_{{}^a\phi_b(q)}$ is the local linearization of ${}^a\phi_b$ at posture q. Iff J_ϕ has full rank ($\operatorname{coldim}(J) = \operatorname{rank}(J)$) each degree of freedom in ${}^a\phi_b(q)$ can be served. However, $J_{{}^a\phi_b(q)}$ is dependent on q and its base vectors can become linear dependent for some q. This happens when the robot leaves the *dexterous workspace* and enters the *reachable workspace*. An example might be a fully stretched human arm (including the index finger): The arm (and finger) cannot be fully stretched while having the index finger pointing back to the shoulder. This translates to the incapability of rotating the finger arbitrarily when the arm is fully stretched.
In general workspaces can be classified by their respective degree:

$$\deg({}^a\mathcal{W}_b) = \operatorname{rank} J_{{}^a\phi_b(q)} - \operatorname{coldim}(J_{{}^a\phi_b(q)}) \tag{2.8}$$

- The *dexterous workspace:* is the space where all degrees of freedom of ${}^a\phi_b(q)$ can be served by the chain between effectors a and b. The dexterous workspace has the degree 0.
- The *reachable workspace:* is the extended space where only a reduced set of the degrees of freedom of ${}^a\phi_b(q)$ can be served. The reachable workspace can be subdivided into subspaces with decreasing degrees until no degree of freedom of ${}^a\phi_b(q)$ can be served. Therefore, the reachable workspaces are of degree -1 to $-n$.

The joint set of all reachable workspaces of effector b with respect to effector a is denoted as:

$${}^a\mathcal{R}_b = \left\{ {}^a\phi(q)_b | \forall q \right\} \tag{2.9}$$

The dimension of ${}^a\mathcal{R}_b$ is the degrees of freedom of ${}^a\phi_b$. For robots in 3 dimensional space this translates to 6 degrees of freedom and 3 degrees of freedom for planar robots[1]. ${}^a\mathcal{R}_b$ can be split into subspaces with different properties regarding each $J_{{}^a\phi_b(q)}$. The *dexterous workspace* ${}^a\mathcal{W}_b$ is a subset of ${}^a\mathcal{R}_b$. Within the dexterous workspace the degree-of-freedom matrix $J_{{}^a\phi_b(q)}$ has full rank.

$${}^a\mathcal{W}_b = \left\{ {}^a\phi(q)_b | \operatorname{coldim}(J_{{}^a\phi_b(q)}) = \operatorname{rank}(J_{{}^a\phi_b(q)}) \right\} \tag{2.10}$$

When the kinematic chain between two effectors a and b contains redundant effectors – joints whose columns in $J_{{}^a\phi_b(q)}$ are linear dependent – then $\mathcal{W}$ contains duplicate instances of the same ${}^a\phi_b$.
This applies to all:

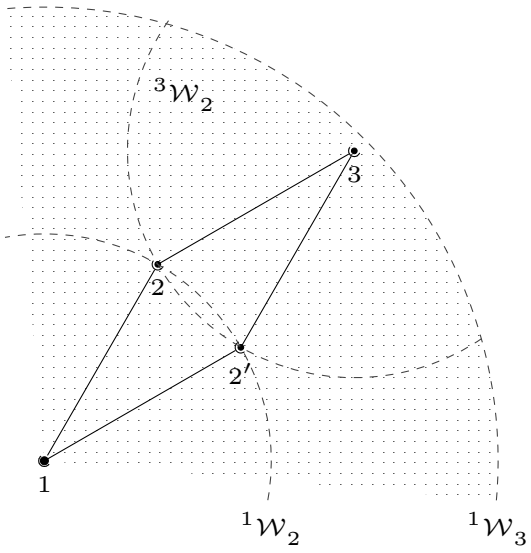
$${}^a\phi_b(q_1) = {}^a\phi_b(q_2) \tag{2.11}$$

[1] The workspace of 1 dimensional (linear) robots has one degree of freedom.

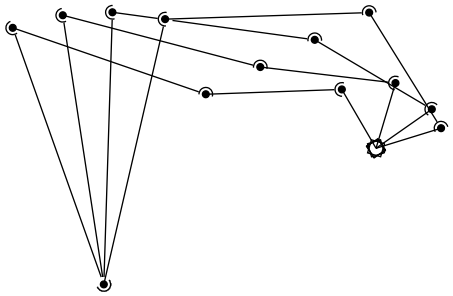
where:

$$q_1 \neq q_2 \text{ and}$$
$${}^a\phi_b(q_1) \in {}^a\mathcal{W}_b \text{ and}$$
$${}^a\phi_b(q_2) \in {}^a\mathcal{W}_b$$

For all q where (2.11) is fulfilled the system has redundant effectors. Thus some effector locations and orientations can be reached with multiple postures. Figure 2.5a displays such a configuration and the workspaces of the robots effectors. Effector 3 can be placed at a single location with two different configurations. The fact that effector configurations can be reached by either a finite – for *redundant* systems – or infinite number – for *hyper redundant* systems – of joint configurations renders inverse kinematics into a nontrivial problem. A hyper redundant system is shown in figure 2.5b.



(a) Redundant configuration of a three-degree-of-freedom system (there is a revolute joint in 3 without a lever) and the workspaces $\mathcal{W}$



(b) Hyper-redundant robot with several postures where the finger resides on the same location.

Figure 2.5: Redundant systems

In figure 2.5a ${}^1\mathcal{W}_2$ and ${}^3\mathcal{W}_2$ are circles but ${}^1\mathcal{W}_3$ is an circular area – the dotted area. At the boundary of ${}^1\mathcal{W}_3$ – where the arm is fully stretched – $\text{rank}(J_{1_{\phi_3(q)}}) = 2$ while inside the dotted area (not at the border) $\text{rank}(J_{1_{\phi_3(q)}}) = 3$. This shows that inside the dotted area the robot is in the *dexterous workspace* (degree 0) and the boundary of this area is the *reachable workspace* (degree -1).

3 - Inverse Kinematics

Inverse kinematics is the answer to the question:

Given a robot's joint configuration and a target state:
How must the robot's posture be changed to achieve the target state?

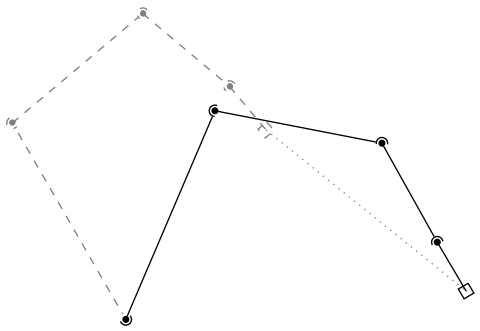


Figure 3.1: Inverse kinematics exemplified

Figure 3.1 illustrates an inverse kinematics problem. The target state in the example is the position of the robot's finger – the box at the end of the kinematic chain. The dashed posture is the initial posture – where the finger is far away from the target state (denoted with the dotted vector) – and the solidly drawn posture is where the robots finger reached the target location. Target states can be any property of the robot like positions, rotations or even more complex configurations. However, inverse kinematics is highly dependent on the robot's structure. As seen in section 2.4 solutions to the general inverse kinematic problem are usually not trivial due to the ambiguity of some elements in the workspace. This chapter will show approaches to solve the inverse kinematics problem.

3.1 Related Work

In real world robot applications there are almost as many inverse kinematic approaches as there are research groups working on robots. An overview about inverse methods is given by Waldron et al. [4]. They subdivided methods to solve the inverse kinematics problem into the following:

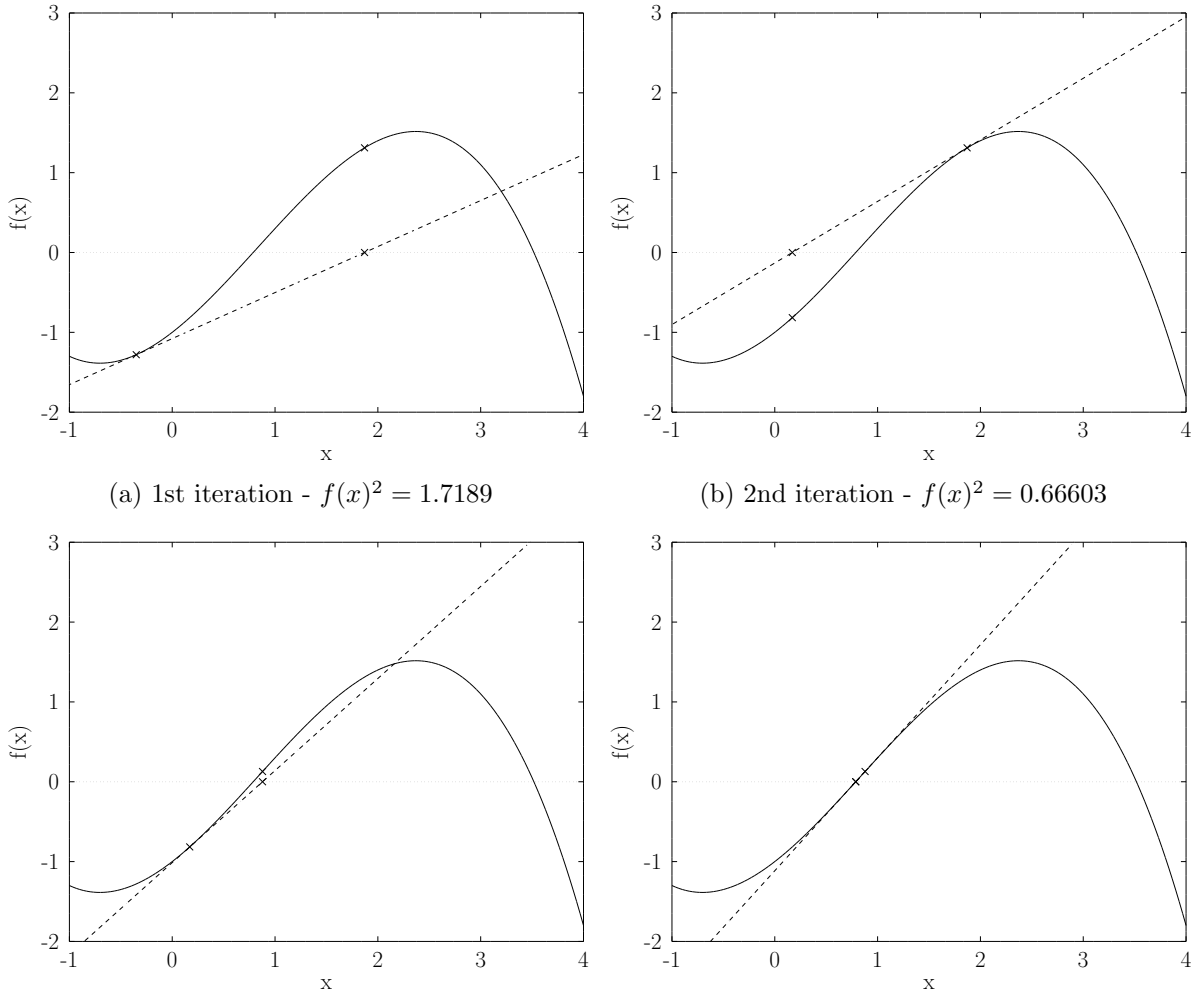
- Closed-form methods:
  The inverse of $\phi(q)$ is expressed as an equation system which can be solved either by inverse trigonometric functions or by dividing the problem along the kinematic path into subproblems which can be solved independently. Closed-form solutions do not exist for arbitrary robots. They can be further divided into two subclasses:
  - Algebraic
  - Geometric
- Numerical methods:
  In an iterative process the inverse of $\phi(q)$ is approximated. For each step q is adjusted to bring the target effector closer to ϕ_{target}. Waldron et al. divide numerical methods into three subclasses:
  - Symbolic elimination
  - Continuation
  - Iterative methods

Closed-form methods are advantageous when the robot has a serial chain structure with up to six degrees of freedom and when the end effector does not leave the *dexterous workspace* [3, 5]. In this case the robot has the same amount of freedom as ϕ which renders the inverse kinematics problem analytically solvable. With this any solution (if there is any) can be found in a fixed time frame. Even the nonexistence of a solution – e.g., when the target state is outside $\mathcal{W}$ – can be found in that fixed time frame. When working on arbitrary robots – especially robots with many redundant effectors – closed-form methods are *not* applicable [5]. Contrary, numerical methods can be utilized for arbitrary robots but have the disadvantage that they need to converge and that the nonexistence of a solution cannot be determined in general. They also do not guarantee to converge on a solution – even if a solution exists. The fact that analytical methods cannot be generalized to arbitrary robots does not outweigh the disadvantages of numerical methods. This renders analytical methods inferior to numerical methods.
The goal of this master's thesis is to derive an inverse kinematics approach for arbitrary robots. The foundation to the approach discussed here was outlined by Donald Lee Pieper in 1968 [5]. In his Ph.D thesis several approaches to generate motion for different types of robots have been introduced including analytical as well as numerical techniques. He also discussed *Newton-Raphson* techniques to approximate inverse kinematics on multiple hyper redundant robotic systems and this technique's performance. His work can be seen as the foundation to most modern inverse kinematics approaches.

3.2 Newton-Raphson

The idea behind the *Newton-Raphson* technique is to iteratively approximate the zero-crossing of a function f. f can be any function; even functions where zero-crossings cannot be calculated directly. As an example in terms of inverse kinematics f can be the forward kinematics of a robot's finger minus the target position. The posture for which $f = 0$ is the posture where the finger reached the target.
Starting at an arbitrary x_0 one calculates the function's local derivative at x_0: $\frac{\delta}{\delta x} f(x_0)$ and value $f(x_0)$ yielding the local tangent. This is also referred to as *local linearization* of f at x.

The process continues with x_1 at the zero-crossing of the tangent derived at x_0. Those steps are repeated until the error $f(x_i)^2$ is less than a threshold. Figure 3.2 displays four iterations of this process.



(a) 1st iteration - $f(x)^2 = 1.7189$

(b) 2nd iteration - $f(x)^2 = 0.66603$

(c) 3rd iteration - $f(x)^2 = 0.016479$

(d) 4th iteration - $f(x)^2 = 5.2229 * 10^{-9}$

Figure 3.2: Newton-Raphson gradient descent to find the zero-crossing of a function

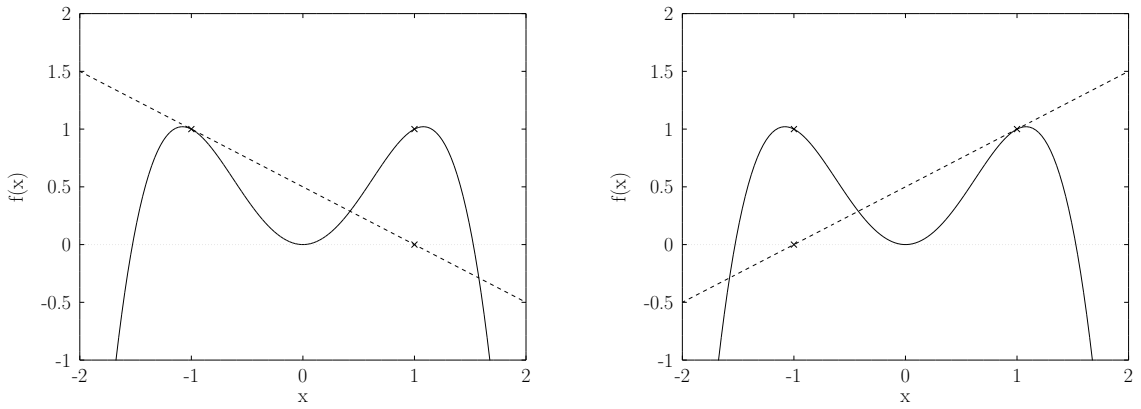
The tangent $t_{x_i}(x)$ at $f(x_i)$ and the tangent's zero-crossing x_{i+1} can be calculated with:

$$t_{x_i}(x) = \overbrace{\frac{\delta}{\delta x} f(x_i)}^{\text{slope}} x + \overbrace{f(x_i) - \frac{\delta}{\delta x} f(x_i) x_i}^{y\text{-offset}} \tag{3.1}$$

$$x_{i+1} = \frac{\frac{\delta}{\delta x} f(x_i) x_i - f(x_i)}{\frac{\delta}{\delta x} f(x_i)} \tag{3.2}$$

It is important to acknowledge that at local minima of f the tangent's slope might become 0. Therefore, the zero-crossing of the tangent would be at infinity. This is especially problematic when the function does not have a zero-crossing at all. Up to a certain iteration the gradient descent would find x_{i+1} with $f(x_{i+1}) < f(x_i)$. But eventually f's slope at some x_{i+n} is less

than f's value at x_{i+n} which results in poor performance in terms of convergence. However, if f is a *convex* function and has at least one zero-crossing then *Newton-Raphson* is guaranteed to converge on one of the zero-crossings. Figure 3.3 displays a situation with a non-convex function where *Newton-Raphson* does not converge but oscillates around the actual zero crossing.



(a) 1st iteration - $f(x)^2 = 1$

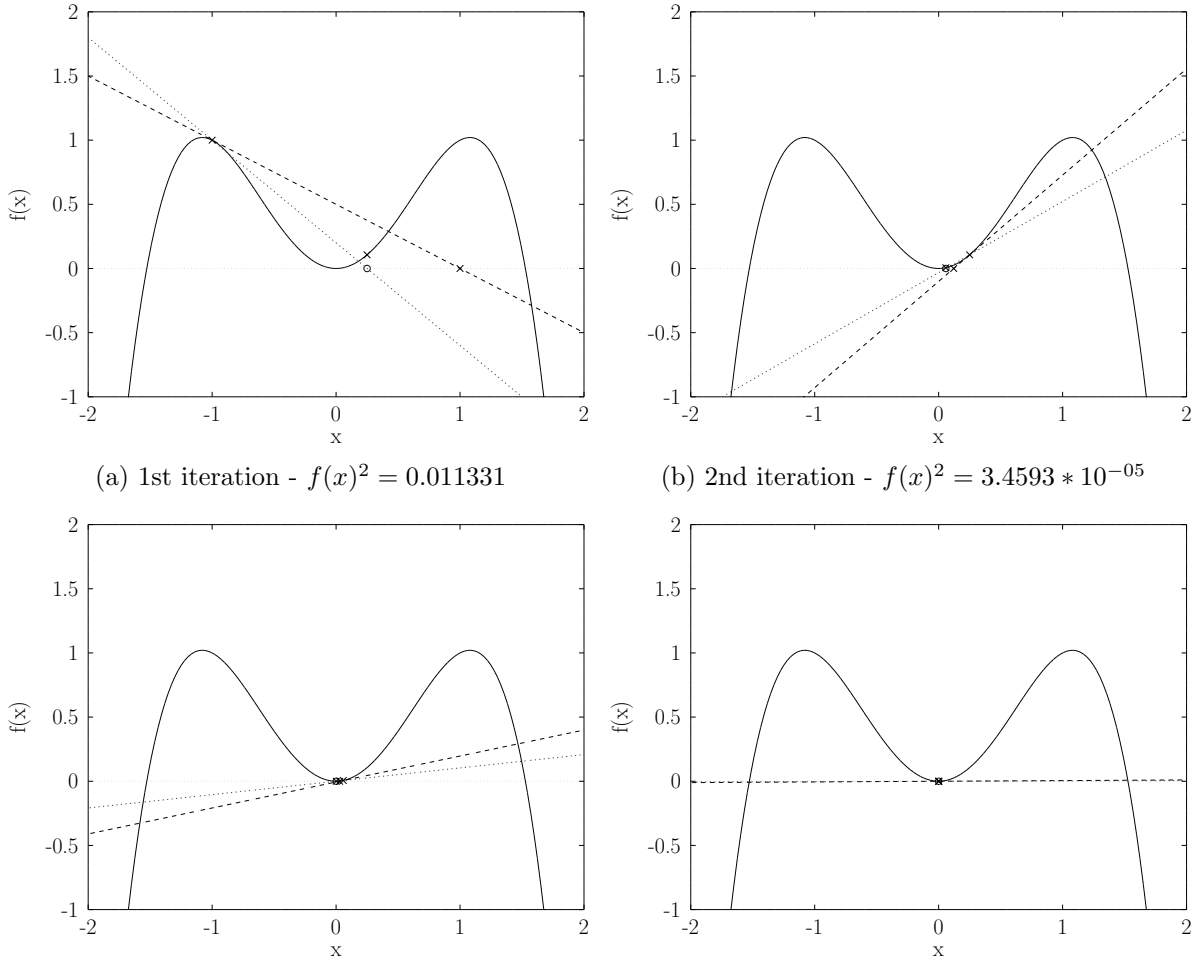
(b) 2nd iteration - $f(x)^2 = 1$

Figure 3.3: A non-convex function where the estimated zero crossing cannot be found but oscillates around the actual zero crossing

To mitigate poor rates of convergence equation (3.2) can be extended to include a *dampening* term. This extension is also referred to as *dampened Newton-Raphson* technique:

$$x_{i+1} = \frac{\left(\frac{\delta}{\delta x} f(x_i) x_i - f(x_i)\right) \frac{\delta}{\delta x} f(x_i)}{\frac{\delta}{\delta x} f^2(x_i) + \epsilon} \tag{3.3}$$

ϵ is the dampening factor. It is easy to see when ϵ approaches 0 the dampened gradient descent becomes the original *Newton-Raphson*. Figure 3.4 shows the same function and starting condition as Figure 3.3 but uses the dampened *Newton-Raphson*. The dampened version does not oscillate and converges quickly to f's zero crossing.



(a) 1st iteration - $f(x)^2 = 0.011331$

(b) 2nd iteration - $f(x)^2 = 3.4593 * 10^{-05}$

(c) 3rd iteration - $f(x)^2 = 1.5551 * 10^{-11}$

(d) 4th iteration - $e^2 = 1.3324 * 10^{-28}$

Figure 3.4: Dampened *Newton-Raphson* descent with $\epsilon = 0.75$

In figure 3.4 the dashed line represents the tangent for the current iteration whereas the fine dashed line marks the *dampened* tangent. The error is calculated as the square of the function's value at the dampened tangent's zero-crossing.
In the next section a loss function will be introduced. The process of finding the zero-crossing of this loss function – that is the location where the error is 0 – is implemented following the pattern described in this section.

3.3 Derivation

Even though inverse kinematics is broadly researched a conclusive derivation that proves optimality of commonly used equations is seldomly shown [6]. This is remarkable as derivations of the *Newton-Raphson* technique are the most dominant inverse kinematics approaches utilized [7]. The goal of this section is to close this gap by providing a conclusive step-by-step derivation of the most commonly utilized dampened least-squares inverse kinematics approach [3, 6–22]. Also, with the derivation a foundation of optimality assertions can be laid out. This section follows mostly the lecture held by Toussaint [23] on robotics where a derivation was outlined

similarly as indicated by Gienger et al. [24].
The process of approximating an effector to a target state – for instance, move the robot's finger to a specific location – can be considered a *task*. A task is defined within the perspective of an effector (the base effector) and contains information about the involved effectors to be solved, a function ψ which maps the robot's configuration q to a state y as well as the target state y^*.

$$\psi(q) \to y \tag{3.4}$$

Task types might be:

- Location:
  Move an effector to a position with respect to another body part. The involved effectors are all effectors along the *path* between the *base effector* and the task's *end effector*. $\psi(q)$ is the translation part of $\phi(q)$.
- Orientation:
  Align the orientation of an effector with a given rotation. The involved effectors are also along the *path* between the *base effector* and the *end effector*. Here $\psi(q)$ is the rotation part of $\phi(q)$.
- Center of mass:
  Move the center of mass to a location as seen from the *base effector*. All effectors of the robot are involved in this task. In this case $\psi(q)$ maps the robot's configuration to the location of the center of mass. Note center of mass tasks do not have a real end effector but uses a property of the whole robot instead.
- Configuration:
  Move an effector i to a defined value q_{it}.
  For this task type the state mapping function is:
  $\psi(q) = q_i$

As shown earlier each task would be easily solvable if ψ^{-1} would exist. However, for arbitrary robots and tasks this is *not* the case [18]. Instead of looking for ways how ψ can be inverted it is possible to reformulate the inverse kinematics problem as a least-squares optimization problem that can be minimized using the *Newton-Raphson* descent technique introduced in section 3.2. To solve a task the robot has to adjust its posture in a way that the target state is reached – or at least approximated. The change of the posture is denoted as Δq. The difference between the target state y^* and the current state is the error $\vec{e} = y^* - \psi(q)$. The change in posture shall be as little as possible while the target state should be reached. This can be rewritten as a square loss function $\mathcal{L}$:

$$\mathcal{L}(\Delta q) = ||y^* - \psi(q + \Delta q)||_C^2 + ||\Delta q||_W^2 \tag{3.5}$$

The notation $||a||_B^2$ represents the square norm of a with the weighting matrix B:

$$||a||_B^2 = a^T B a$$

C weights the importance in task space. Since the solution of the task is of paramount importance C is set to ∞. W weights the usage of active effectors to solve the task. Since C clearly dominates in (3.5) W can be set to I. If movements on specific effectors shall be suppressed the corresponding main diagonal entry in W can be set to a high value.

The $\Delta\hat{q}$ minimizing $\mathcal{L}$ is the solution of the task:

$$\Delta\hat{q} = \underset{\Delta\hat{q}}{\operatorname{argmin}}\, \mathcal{L}(\Delta\hat{q}) \tag{3.6}$$

As long as y^* is reachable in terms of ψ the loss function suffices the optimality criterion of the *Newton-Raphson* gradient descent and thus is solvable.
Analogous to the *Newton-Raphson* technique (3.3) $\Delta\hat{q}$ (in (3.6)) can be found by iteratively approximating and accumulating Δq. To achieve this, $\mathcal{L}$ has to be derived with respect to Δq and set to zero. Since the state mapping function $\psi(q)$ is *not* trivially invertible – especially not for arbitrary robots – it is necessary has to use the locally linearized derivative of $\psi(q)$ at q [1]:

$$\lim_{\Delta q \to 0} \psi(q + \Delta q) = \psi(q) + J\Delta q \tag{3.7}$$

Here J is the task's *Jacobian* of ψ at q. Note that q is a parameter of J. When implementing this inverse kinematics solver J has to be recomputed each time q changes.

$$\begin{aligned}
\frac{\delta}{\delta\Delta q}\mathcal{L}(\Delta q) = 0 &= \frac{\delta}{\delta\Delta q}\left[||J\Delta q - \vec{e}||_C^2 + ||\Delta q||_W^2\right] \\
&= \frac{\delta}{\delta\Delta q}\left[(J\Delta q - \vec{e})^T C (J\Delta q - \vec{e}) + \Delta q^T W \Delta q\right] \\
&= \frac{\delta}{\delta\Delta q}\left[(\Delta q^T J^T - \vec{e}^T) C (J\Delta q - \vec{e}) + \Delta q^T W \Delta q\right] \\
&= \frac{\delta}{\delta\Delta q}\left[\Delta q^T J^T C J \Delta q - 2\vec{e}^T C J \Delta q + \Delta q^T W \Delta q\right] \\
&= 2J^T C^T J\Delta q - 2J^T C^T \vec{e} + 2W^T \Delta q \\
&= J^T C^T J\Delta q - J^T C^T \vec{e} + W^T \Delta q \\
&= (J^T C^T J + W^T)\Delta q - J^T C^T \vec{e} \\
J^T C^T \vec{e} &= (J^T C^T J + W^T)\Delta q \\
\Delta q &= (J^T C^T J + W^T)^{-1} J^T C^T \vec{e} \qquad (3.8) \\
\boxed{\Delta q = W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1} \vec{e}} \qquad (3.9)
\end{aligned}$$

(3.8) is the preliminary solution of the derivation. Due to $C \to \infty$ (3.8) is not numerically utilizable. To transform (3.8) to (3.9) Toussaint [23] suggested to apply the Woodbury-Identity[25]. However, by utilizing the Woodbury-Identity one does *not* get (3.9). Yet it can be directly shown that the equations (3.8) and (3.9) have to be equivalent:

$$\begin{aligned}
(J^T C^T J + W^T)^{-1} J^T C^T &= W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1} \qquad (3.10) \\
J^T C^T &= (J^T C^T J + W^T) W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1} \\
J^T C^T (J W^{T^{-1}} J^T + C^{T^{-1}}) &= (J^T C^T J + W^T) W^{T^{-1}} J^T \\
J^T C^T J W^{T^{-1}} J^T + J^T C^T C^{T^{-1}} &= J^T C^T J W^{T^{-1}} J^T + W^T W^{T^{-1}} J^T \\
J^T C^T J W^{T^{-1}} J^T + J^T &= J^T C^T J W^{T^{-1}} J^T + J^T \qquad (3.11)
\end{aligned}$$

[1] This is the equivalent of the slope of the tangent introduced in equation (3.1).

□

Continuing with (3.9):

$$\Delta q = W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1} \vec{e}$$

With $\lim_{C \to \infty} C^{T^{-1}} = \lim_{\epsilon \to 0} \epsilon I$ and $W = I$:

$$\Delta q = \lim_{\epsilon \to 0} J^T (J J^T + \epsilon I)^{-1} \vec{e} \tag{3.12}$$

$$\boxed{\Delta q = J^\dagger \vec{e}} \tag{3.13}$$

Here $J^\dagger$ denotes the *Moore-Penrose pseudoinverse* [26] of J. Note the similarity between (3.12) and the dampened *Newton-Raphson* (3.3). The *dampening* factor $\epsilon I \neq 0$ guarantees that $(J^T J + \epsilon I)$ is invertible since $J^T J$ must be positive semidefinite and symmetric.
It is important to acknowledge that $J^T (J^T J + \epsilon I)^{-1}$ is *not* the "real" pseudoinverse of J but approaches $J^\dagger$ for $\epsilon \to 0$. In iterative inverse kinematics implementations a nonzero ϵ should be chosen to avoid singularities where J does not have full rank and to dampen the iterative solution process. Otherwise the iterations might jump around the actual solution due to the non-convex nature of ψ where the local linearization produces overshooting Δqs (see 3.3).
It is worth pointing out that the identity (3.10) is useful when dealing with least-squares optimization processes that are in a form as in (3.8) and therefore numerically not solvable. That is especially the case when a W^{-1} has to be used which does not have full rank. (3.10) can turn those equations into a solvable form. This property will be utilized in chapter 5.

3.4 Moore-Penrose Pseudoinverses

Pseudoinverses are a generalization of inverses for non-square full rank matrices [26]. Any nonsingular matrix A has a unique pseudoinverse with the four properties:

$$A A^\dagger A = A \tag{3.14}$$

$$A^\dagger A A^\dagger = A^\dagger \tag{3.15}$$

$$(A A^\dagger)^T = A A^\dagger \tag{3.16}$$

$$(A^\dagger A)^T = A^\dagger A \tag{3.17}$$

If A is square and has full rank A's pseudoinverse is its inverse:

$$A^\dagger = A^{-1} \tag{3.18}$$

The equations (3.16) and (3.17) show that any matrix multiplied with its pseudoinverse yields a Hermitian matrix.
(3.14) can be expanded to:

$$\begin{aligned} A A^\dagger A &= A \\ A A^\dagger A &= A A^\dagger A A^\dagger A \\ A A^\dagger &= A A^\dagger A A^\dagger \end{aligned} \tag{3.19}$$

and analogous for (3.15):

$$A^\dagger A = A^\dagger A A^\dagger A \tag{3.20}$$

It becomes clear that the matrices $AA^\dagger$ and $A^\dagger A$ are idempotent. Here $A^\dagger A$ is the orthogonal *range projector* of the row space of A – further denoted as $\mathcal{M}$ – and $AA^\dagger$ is the orthogonal range projector of the column space of A. This means any vector multiplied to the right hand side to $AA^\dagger$ or $A^\dagger A$ is projected orthogonally into the respective rangespace. The complement of the *range projector* is the *nullspace projector* $\mathcal{N}$:

$$\mathcal{N}_{Arow} = I - AA^\dagger \tag{3.21}$$
$$\mathcal{N}_{Acol} = I - A^\dagger A \tag{3.22}$$

Any vector $\vec{v}$ multiplied to the right hand side of $\mathcal{N}$ is projected into the space that is *not* spanned by either the columns or rows of A. Proof by showing the range of A is orthogonal to the range of $\mathcal{N}$:

$$\begin{aligned} 0 &= A\mathcal{N}_{Acol} \\ &= A(I - A^\dagger A) \\ &= A - AA^\dagger A \qquad \text{apply (3.14)} \\ &= A - A \end{aligned}$$

□

analogous for $\mathcal{N}_{Acol}$:

$$\begin{aligned} 0 &= \mathcal{N}_{Acol}A \\ &= (I - AA^\dagger)A \\ &= A - AA^\dagger A \qquad \text{apply (3.14)} \\ &= A - A \end{aligned}$$

□

The *dampened* Moore-Penrose pseudoinverses used in (3.12) overcome the constraint that A has to be of full rank on the cost of accuracy. But as shown in section 3.2 this dampening is in fact an intended feature. A different – and numerically more stable – way to build the pseudoinverse is by using the *singular value decomposition* of J [11, 18, 27–29]:

$$J \in \mathcal{R}^{n \times m}$$
$$J = U\Sigma V^T \tag{3.23}$$

with:

$$U \in \mathcal{R}^{n \times n}$$
$$\Sigma \in \mathcal{R}^{n \times m}$$
$$V \in \mathcal{R}^{m \times m}$$
$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \sigma_n \end{pmatrix}$$

U and V are orthogonal matrices and Σ is a diagonal matrix with J's singular values on the main diagonal – in descending order. Using the singular value decomposition $J^\dagger$ can be constructed by:

$$J^\dagger = V\Sigma^\dagger U^T$$

with:

$$\Sigma^\dagger \in \mathcal{R}^{m\times n}$$
$$\Sigma^\dagger = \lim_{\epsilon\to 0} \begin{pmatrix} \frac{\sigma_1}{{\sigma_1}^2+\epsilon} & 0 & \dots & 0 \\ 0 & \frac{\sigma_2}{{\sigma_2}^2+\epsilon} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \frac{\sigma_n}{{\sigma_n}^2+\epsilon} \end{pmatrix} \tag{3.24}$$

Proof:

$$\begin{aligned} J^\dagger &= J^T(JJ^T)^{-1} \\ &= (U\Sigma V^T)^T((U\Sigma V^T)(U\Sigma V^T)^T)^{-1} \\ &= V\Sigma^T U^T(U\Sigma V^T V\Sigma^T U^T)^{-1} \\ &= V\Sigma^T U^T(U\Sigma\Sigma^T U^T)^{-1} \\ &= V\Sigma^T U^T {U^T}^{-1}(\Sigma\Sigma^T)^{-1}U^{-1} \\ &= V\Sigma^T(\Sigma\Sigma^T)^{-1}U^{-1} \\ &= V\Sigma^\dagger U^T \end{aligned}$$

□

In equation (3.24) ϵ is the dampening factor and chosen similar to the ϵ in the previously introduced dampened pseudoinverse. Usually ϵ is chosen to be nearly 0 to approximate the "real" pseudoinverse and still have some dampening.
The rangespace projector $\mathcal{M}$ can also be calculated using the singular value decomposition:

$$\begin{aligned} \mathcal{M} = AA^\dagger &= U\Sigma V^T V\Sigma^\dagger U^T \\ &= U\Sigma\Sigma^\dagger U^T \end{aligned} \tag{3.25}$$

Because U is an orthonormal matrix and Σ as well as $\Sigma^\dagger$ are diagonal matrices where the entries along the main diagonal are either σ_i and $\frac{1}{\sigma_i}$ or 0 respectively (see (3.24)) the resultant range projector of A is a matrix with the singular values of either 0 or 1. This also means that the eigenvalues of the range projector (and the nullspace projector) are also either 0 or 1. Furthermore, the determinant of UU^T has to be 1 whereas the determinant of $\Sigma\Sigma^\dagger$ is either 0 or 1. Therefore, the determinant of $AA^\dagger$ is either 1 or 0 as well. This translates to A having full rank or not [13].
Furthermore, because $UU^T = I$:

$$\text{trace}(U\Sigma\Sigma^\dagger U^T) = \text{trace}(\Sigma\Sigma^\dagger)$$

and because $\sigma_i {\sigma_i}^\dagger \in 0, 1$:

$$\begin{aligned} \text{trace}(U\Sigma\Sigma^\dagger U^T) &= \text{rank}(\Sigma\Sigma^\dagger) \\ \text{trace}(U\Sigma\Sigma^\dagger U^T) &= \text{rank}(AA^\dagger) \end{aligned}$$

and because of $\text{rank}(AA^\dagger) = \text{rank}(A)$:

$$\begin{aligned} \text{trace}(U\Sigma\Sigma^\dagger U^T) &= \text{rank}(A) \\ \text{trace}(\mathcal{M}) &= \text{rank}(A) \end{aligned} \tag{3.26}$$

The same applies to the trace of $A^\dagger A$. This means by calculating the trace of A's range projector one calculates the rank of A as well.
As the singular values approach 0 some base vectors of J become collinear or at least almost collinear. When this happens the rank of J collapses which renders J singular. This is the case at the transition from the dexterous to the reachable workspace and is typically *not* a continuous transition. However, by utilizing the dampened pseudoinverse with $\epsilon \neq 0$ of J the collapsing rank can be made into a smooth transition. Figure 3.5 plots the trace of the range projector of a matrix versus varying ϵ and versus an angle α between the base vectors of the matrix.

$$\text{defect} = \dim(A) - \mathit{trace}({AA^\dagger}_\epsilon)$$

with

$$\begin{aligned} A &= \begin{pmatrix} 1 & \cos\alpha \\ 0 & \sin\alpha \end{pmatrix} \\ {A^\dagger}_\epsilon &= A^T(AA^T + \epsilon I)^{-1} \end{aligned}$$

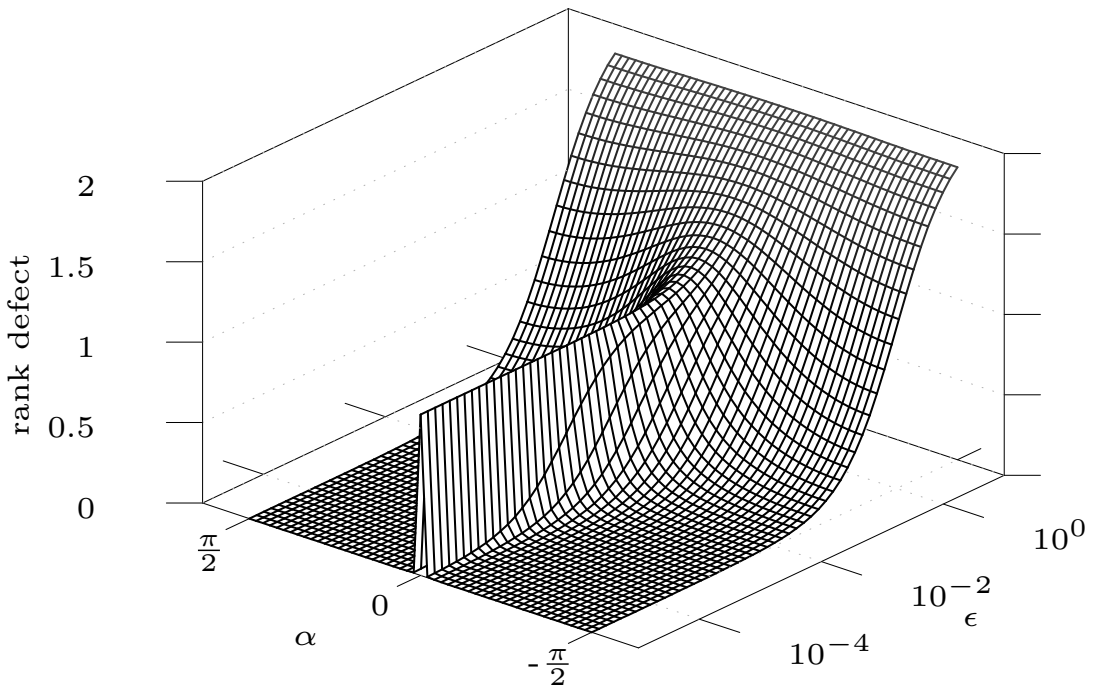


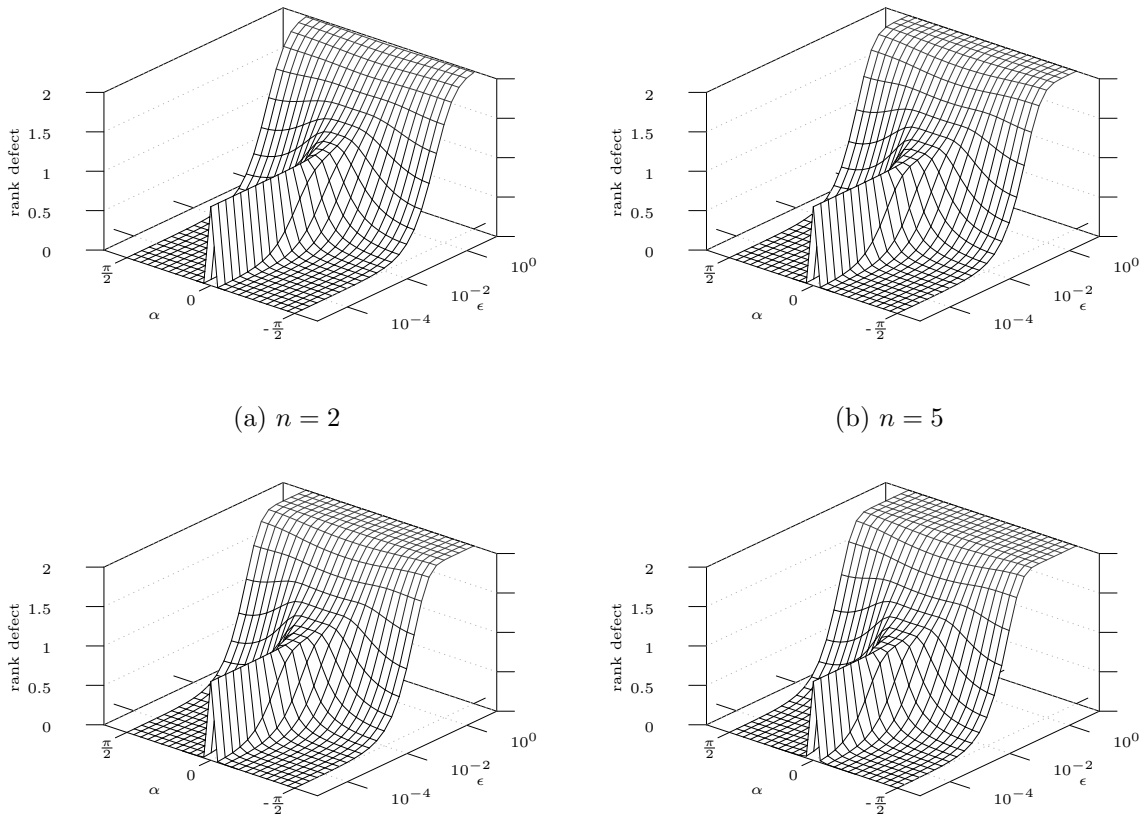
Figure 3.5: Rank defect calculated by using the trace of the dampened range projector

Figure 3.5 shows that for $\alpha \neq 0$ the rank defect is 0 for sufficiently small ϵ. It also serves as a hint on how to pick "good" values for ϵ as the height of the plot corresponds to the total dampening in configuration space when utilizing the dampened pseudoinverse of a task's Jacobian. Even when a task's Jacobian has full rank some dampening might be preferred to avoid overshooting but for the sake of accuracy and quick convergence ϵ should be chosen as small as possible. However, at near singular configurations $\text{trace}(J) \to \dim(J) - n$ it might be preferable to have additional dampening.
As indicated above, the rank defect happens at the transition between a workspace of degree $-n$ to a different degree. By using the dampened range projector the transition can be detected and the degrees of freedom that are about to become defective are in the nullspace projector of J^T. The detection of this transition can be utilized to prevent a task from moving further in the direction where the rank loss would happen or to add an additional dampening of movements near the border to any reachable space.
Furthermore, due to $\epsilon \neq 0$ the dampened nullspace projector $\mathcal{N}_d$ is actually *not* idempotent. $\mathcal{N}_d$ contains residual singular values that reflect ϵ as well as numerical inaccuracy. This can be utilized to further modify the dampening behavior when in near singular configurations:

$$\text{defect} = \dim(A) - trace((AA^{\dagger}{}_{\epsilon})^n) \tag{3.27}$$

The exponent n creates a plateau in near singular configurations. In figure 3.6 this plateau and its behavior with varying ϵ and exponents is displayed. The greater the exponent n the more

distinct the plateau becomes. This behavior will be used to improve numerical stability for the utilization of nullspace projectors in chapter 5.



(a) $n = 2$

(b) $n = 5$

(c) $n = 10$

(d) $n = 20$

Figure 3.6: Rank defect calculated by using the trace of the exponentiated dampened range projector

4 - Jacobians

This chapter explains the derivation of the local linearization of $\psi(q)$ used in (3.7). $\psi(q)$ is the function that maps the robot's configuration to the *task space* (some examples are shown in 3.1). Figure 4.1 displays an example where $\psi(q)$ is the location of a robot's finger. In this example the locally linearized derivative of ψ at q is a matrix where each column vector represents the effect the corresponding effector to the location on the robot's finger.
The *column space* of J spans the d-dimensional *task space* and the row space the *configuration space* of the task. The number of columns of J is therefore the same as the number of active effectors of the robot. The i-th column represents the i-th active effector. For all tasks of a specific robot the number and mapping of the columns of J is fixed. The number of rows, however, is dependent on the task. E.g., for 2-dimensional location tasks J has two rows; three rows for 3-dimensional location tasks.
The general form of the Jacobian is:

$$J = \frac{\delta}{\delta q}\psi(q) = \begin{pmatrix} \frac{\delta}{\delta q_1}\psi(q)_1 & \frac{\delta}{\delta q_2}\psi(q)_1 & \cdots & \frac{\delta}{\delta q_n}\psi(q)_1 \\ \frac{\delta}{\delta q_1}\psi(q)_2 & \frac{\delta}{\delta q_2}\psi(q)_2 & \cdots & \frac{\delta}{\delta q_n}\psi(q)_2 \\ \vdots & \vdots & \ddots & \vdots \\ \underbrace{\frac{\delta}{\delta q_1}\psi(q)_d}_{*1} & \underbrace{\frac{\delta}{\delta q_2}\psi(q)_d}_{*2} & \cdots & \underbrace{\frac{\delta}{\delta q_n}\psi(q)_d}_{*3} \end{pmatrix} \tag{4.1}$$

*1: effect of the first active effector on $\psi(q)$

*2: effect of the second active effector on $\psi(q)$

*3: effect of the n-th active effector on $\psi(q)$

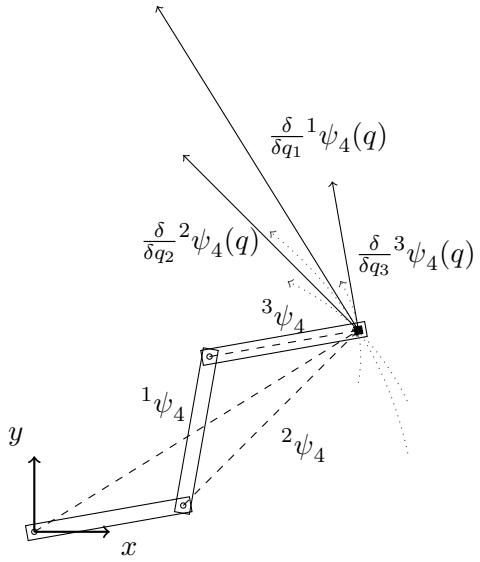


Figure 4.1: Visualization of a Jacobian matrix

The Jacobian can also be interpreted as the translation between *configuration-velocity* $\dot{q} = \frac{d}{dt}q$ and *task-velocity* $\dot{\psi}(q) = \frac{d}{dt}\psi(q)$ at posture q [2]:

$$\dot{\psi}(q) = J\dot{q} \tag{4.2}$$

When recalling equation (2.7) it becomes clear that $J_{^a\phi_b(q)}$ is the combined Jacobian for location and rotation.

$$J_{^a\phi_b(q)} = \begin{pmatrix} \frac{\delta}{\delta q}{}^a\phi_{bx}(q) \\ \frac{\delta}{\delta q}{}^a\phi_{by}(q) \\ \frac{\delta}{\delta q}{}^a\phi_{bz}(q) \\ \frac{\delta}{\delta q}{}^a\phi_{br1}(q) \\ \frac{\delta}{\delta q}{}^a\phi_{br2}(q) \\ \frac{\delta}{\delta q}{}^a\phi_{br3}(q) \end{pmatrix} \tag{4.3}$$

This also emphasizes that workspaces are in fact task specific.
The index i at ${}^a\phi_{bi}$ denotes the i-th degree of freedom of $\phi(q)$.

Figure 4.1 shows an example where the task manipulates the 2D location of a robot's finger and the task space speed is the finger's velocity vector. The figure also shows the column vectors of J and the vectors ${}^a\psi_4$ between each joint and the *end effector* (4) as well as the arcs where the *end effector* moves when solely the corresponding joint is changed. The partial derivatives in this posture are tangential vectors to the arcs with length of $||{}^a\psi_4||$ along the positive rotation direction of each joint. The fact that the partial derivatives have to be the tangents on the arcs makes it obvious that the partial derivative of a revolute joint i on the effector e is [2]:

$$\frac{\delta}{\delta q_i}{}^i\psi_e(q) = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} {}^i\psi_e(q) \tag{4.4}$$

3D case:

$$\frac{\delta}{\delta q_i}{}^i\psi_e(q) = r_i \times {}^i\psi_e(q) = [r_i]\,{}^i\psi_e(q) \tag{4.5}$$

The $\times$ operator used in (4.7) denotes the cross product between the rotation axis of the i-th joint and the vector from this joint to the end effector and $[r_i]$ denotes the skew-symmetric cross-product matrix of r_i.
Analogous for prismatic joints:

$$\frac{\delta}{\delta q_i}{}^i\psi_e(q) = \begin{pmatrix}\vec{d}_0\end{pmatrix}$$

Here $\vec{d}_0$ is the normed direction vector along which the prismatic joint works.
Generally each joint has a transformation which can be utilized to build the corresponding entry for the Jacobian matrix. This is the locally linearized derivative of the intrinsic transformation at its base configuration ($q_i = 0$) for q: $\frac{\delta}{\delta q_i}T_{i_{int}}(q_i = 0)$
E.g., for two dimensional prismatic joints:

$$\frac{\delta}{\delta q_i}T_{i_{int}} = \frac{\delta}{\delta q_i}\begin{pmatrix}0 & 0 & q_i\vec{d}\\ 0 & 0 & \\ 0 & 0 & 1\end{pmatrix} = \begin{pmatrix}0 & 0 & \vec{d}\\ 0 & 0 & \\ 0 & 0 & 0\end{pmatrix} \tag{4.6}$$

And for revolute joints:

$$\frac{\delta}{\delta q_i}T_{i_{int}} = \frac{\delta}{\delta q_i}\begin{pmatrix}\cos q_i & -\sin q_i & 0\\ \sin q_i & \cos q_i & 0\\ 0 & 0 & 1\end{pmatrix} = \begin{pmatrix}-\sin q_i & -\cos q_i & 0\\ \cos q_i & -\sin q_i & 0\\ 0 & 0 & 1\end{pmatrix} \overset{q_i=0}{=} \begin{pmatrix}0 & -1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 0\end{pmatrix} \tag{4.7}$$

The derivation for revolute joints conveniently generates the skew-symmetric matrix for the three dimensional case used earlier but can be further generalized for higher and lower dimensions. The representation of the transformations in (4.6) and (4.7) are slightly different to the notation above ((4.4) and (4.5)). This is advantageous as they work for any homogeneous coordinate. This notation simplifies the generation of Jacobians because tasks for translations and rotations can be treated equally by using the corresponding column of the forward transformation from each effector to the end effector. All columns in ϕ but the last denote *directions* (the direction of the x, y and z axis) and have a 0 as last element while the last column represents a location which is augmented with a 1. Since the last row of the locally linearized derived transformation is constantly 0 for all joint types and it does not serve any purpose it can be dropped.

4.1 Programmatic Generation of Jacobians

For simple implementation tasks can be divided into specific types:

- *Configuration Tasks* — Tasks that control joints directly:
  Those tasks are independent on a robot's structure since their task space is the configuration space. The *default value task* described in section 4.1.1 in such a task.

- *Pathed Tasks* — Tasks that rely on a robot's structure but work along the path from one effector to another effector:
  Each of those tasks is calculated in a very similar fashion. Examples are tasks that operate on the kinematic chain from the base effector to the end effector. The calculation of Jacobians for this task is described in section 4.1.2.

- *Dynamics Tasks* — Tasks to control physical properties of a robot:
  With this task types dynamic properties like the position of the center of mass or the inertia can be manipulated [1]. The task that controls the position of the center of mass is described in section 4.1.4.

4.1.1 Configuration Tasks

The Jacobian for the default values task is the easiest to implement and does not depend on the robot's structure at all. Here the Jacobian has to fulfill the function: $\psi(q) = q_i$ where i is the joint this task controls. J has only one row because it controls only one degree of freedom of the robot.

$$J = \begin{pmatrix} 0 & \cdots & 0 & \underbrace{1}_{\text{at index } i} & 0 & \cdots 0 \end{pmatrix} \tag{4.8}$$

The error vector for this task type calculates as:

$$\vec{e} = q_{it} - q_i \tag{4.9}$$

q_{it} denotes the joint's target value.

4.1.2 Pathed Tasks

Pathed Tasks manipulate spatial properties of the *kinematic chain* from a base effector a to an end effector b. Properties might be the location or rotation of b as seen from a or the position of the combined mass of the chain. Since there is a single unique path between a and b those tasks can be considered to work along a *path* (as described in 2.3) within the robot. Some effectors might *not* be part of the path thus they cannot manipulate the task space and are reflected with zeros in their respective columns of the Jacobians. Jacobians for pathed tasks are calculated by traversing along the inverse path and at each traversal step filling in the corresponding column in the Jacobian.

4.1.3 Location and Orientation Tasks

For those task types a *path* from the end effector to the base effector is needed. Note this task actually works by utilizing the reversed version of the path that is used to calculate the current value of the task. To calculate the jacobian it is necessary to walk along this reversed path from the end effector to the base effector, accumulate the Jacobian and multiplicatively accumulate the transformations from the current effector to b. The parameter `dimension` is a number between 0 and $\dim(\phi) - 1$ and denotes the column of the forward transformation for which the derivative should be built. E.g. for `dimension = 0` the Jacobian is built that can manipulate the orientation of the x-axis of the end effector with respect to the base effector and for `dimension = dim(`ϕ`)-1` the end effector's location.

[1] Controlling the inertia matrix in terms of inverse kinematics is possible but not part of this thesis.

Listing 4.1: Exemplary Jacobian calculation for 2 dimensional robots

```
def getJacobian(robot, path, dimension):
  transform = np.matrix(np.eye(3, 3))
  jacobian  = np.matrix(np.zeros((2, self.dof)))
  for i in range(0, len(path)):
    node, direction = path[i]
    subtransform = np.matrix(np.eye(3, 3))
     if direction == Direction.FROM_CHILD or \
        direction == Direction.LINK:
      subtransform = path[i-1][0].getTransform()
    elif direction == Direction.FROM_PARENT:
      subtransform = node.getBackTransform()
    transform = subtransform * transform
    jacobian = subtransform[0:2,0:2] * jacobian

    if direction == Direction.FROM_CHILD:
      jacobian[:,node.idx] = node.getDerivative(transform[:,dimension])
    elif direction == Direction.FROM_PARENT:
      jacobian[:,node.idx] = -node.getDerivative(transform[:,dimension])
  return jacobian
```

Algorithm 4.1 starts by initializing the Jacobian with zeros and `transform` with identity. `transform` is a homogeneous transformation that contains the transformation from the current node in the path to the end effector. Each element in `path` contains the node and information about the traversal direction with respect to the structure of the kinematic tree. There are four cases for the direction depending on which the calculations are slightly different:

- `BEGINNING`:
  This is the first node in the path.
- `FROM_CHILD`:
  The previous node in the path is a child of the current node.
- `FROM_PARENT`:
  The previous node in the path is the parent of the current node. Here the node acts inverted to the path's perspective since it moves its parent instead of its children.
- `LINK`:
  When the path contains traversal upwards as well as downwards this marks the node where the turnaround happens. `LINK`-nodes are not part of the active nodes in the path since they cannot move their children independently.

Depending on the direction of traversal the accumulation of the transform from the current effector to the task's end effector is slightly different. However, the way it is accumulated in the algorithm guarantees that `transform` is correct as it follows the same pattern as when calculating the forward transform as depicted in (2.5).
`node` is an instance of the abstract base class `Node`. Subclasses of `Node` implement the method `getDerivative(vec)`. Depending on the node type it is implemented as (4.7) or (4.6). At each iteration of the for-loop the Jacobian is first transformed into the current node's coordinate frame. The transformation used (`subtransform`) is the rotation submatrix of the homogeneous transformation from the previous node to the current node. The translation part of this matrix

does not influence J since derivatives of functions do not reflect offsets – translations would be the equivalent to offsets here.

4.1.4 Dynamics Tasks - Center of Mass

The center of mass (COM) is the equivalent point mass of a multibody structure. The COM has a position – as seen from the base effector – and a mass. The location of the COM is calculated as:

$$p_{COM} = \frac{1}{\sum_i w_i} * \sum_i p_i * w_i \tag{4.10}$$

w_i is the mass of the i-th body and p_i is its location. To calculate the COM the robot's kinematic structure must contain mass information in a fashion where each node contains the equivalent mass attached to it. Because masses cannot be moved with respect to the effector they are attached to, each mass is a static property of the overall robot.
In contrast to tasks controlling locations and orientations the COM-task does not have an end effector which is a part of the robot's body. What acts as the equivalent to the end effector is a virtual effector that depends on the entirety of the robotic system. Therefore, COM-tasks cannot be expressed with *paths* but they can be built recursively. By traversing from the base node along all branches of the tree the (combined) mass each node can manipulate has to be calculated. Depending on whether the current direction of traversal is upwards or downwards the tree nodes can either manipulate the mass that is directly attached to them or not. The base node, however, cannot manipulate the location of the mass that is directly attached to it because it is a fixed property of that node.

Listing 4.2: Jacobian calculation for COM-tasks

```
def getJacobian(robot, baseNode)
  totalMass = Mass()
  jacobian = traverseTree(robot, baseNode, None, totalMass)
  return jacobian * 1 / totalMass.getMass()

def traverseTree(robot, node, prevNode, massFromSubtree):
  jacobian = np.matrix(np.zeros((2, robot.numDOF)))
  for child in node.children:
    if child != prevNode:
      childMass = Mass()
      jacobian += child.getForwardTransform()[0:2,0:2] * \
        traverseTree(robot, child, node, childMass)
      massFromSubtree += \
        childMass.applyTransform(child.getForwardTransform())

  parent = node.parent
  if parent && parent != prevNode:
    parentMass = Mass()
    jacobian += node.getBackwardTransform()[0:2,0:2] * \
      traverseTree(robot, parent, node, parentMass)
    massFromSubtree += \
      parentMass.applyTransform(node.getBackwardTransform())

  if parent != prevNode:
    jacobian[:,node.idx] = \
      -activeNode->getDerivative(massFromSubtree.position) * \
```

```
27        massFromSubtree.getMass()
28      if prevNode:
29      massFromSubtree += node->getMass()
30    else:
31      if prevNode:
32        massFromSubtree += node->getMass()
33        jacobian[:,node.idx] = \
34          activeNode->getDerivative(massFromSubtree.position) * \
35          massFromSubtree.getMass()
36    return jacobian
```

In algorithm 4.2 a class `Mass` is utilized which implements the `+` operator to calculate the combined mass (with the correct position and mass) of the left and right hand side arguments. A useful feature of the Jacobian that describes the center of mass' manipulability is that it creates an easy way to calculate the impulse (linear momentum) of the robot as seen from an effector:

$$\vec{p} = J_{COM} * \dot{q} * \sum_i w_i \tag{4.11}$$

In equation (4.11) i iterates over all masses that can be manipulated.

4.2 Combining Jacobians – Simple Multiple Tasks

When examining Jacobians – specifically their rows – it is noticeable that the orientation of the task space is arbitrary. Each task performs identical for any invertible transformation that is applied on the task space of the Jacobian as well as the error. If the transformation does not have full rank tasks that solve for subspaces of the actual task space can be expressed. To emphasize this equation (4.2) shall be recalled:

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

with any matrix M:

$$M\dot{\psi}(q) = MJ\dot{q} \tag{4.12}$$

M is the task space transformation which is not necessarily a square matrix. M can be any matrix that maps a metric from the untransformed task space to the transformed. An option to utilize M would be to "cut" rows from J to create a task that operates on a subset of the base vectors spanning the original task space. In contrast to the term *degrees of freedom* of the task space the remaining task space is called *degrees of control* of the task space. Analogous to removing degrees of control from the task degrees of control can be added. Therefore, Jacobians can be "stacked" on top to generate a *big Jacobian* $\hat{J}$:

$$\hat{J} = \begin{pmatrix} J_1 \\ J_2 \\ \vdots \\ J_n \end{pmatrix} \tag{4.13}$$

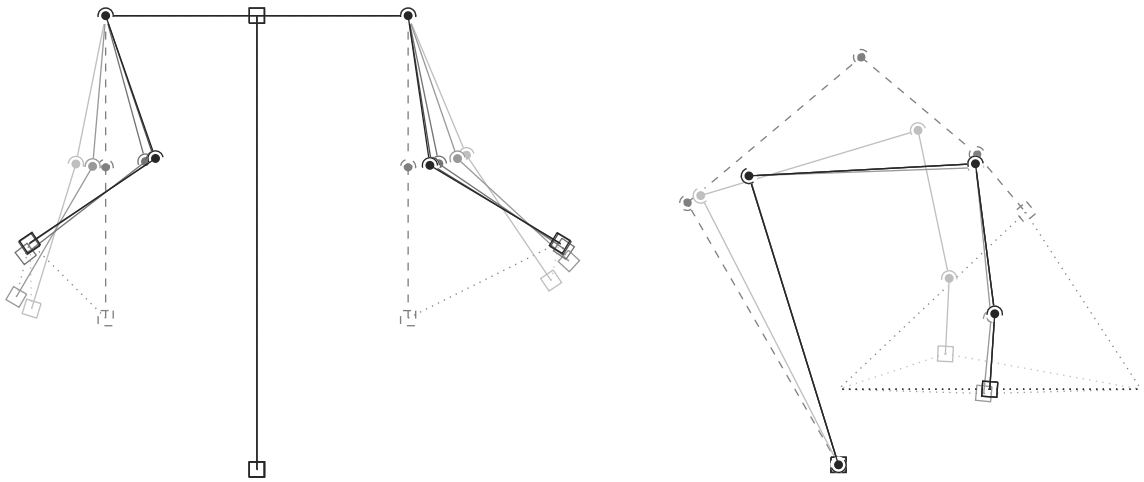
The same has to be done for the error:

$$\hat{e} = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} \tag{4.14}$$

Mansard and others refer to $\hat{J}$ as *augmented Jacobian* [14, 17, 30] or group of tasks. *Augmented Jacobians* act exactly like a single task even though the combined task space might be redundant. Figure 4.2a displays a simple robotic system with two fingers. Here two tasks – one for each finger – are solved simultaneously. When utilizing the augmented Jacobian even conflicting tasks can be formulated. Figure 4.2b visualizes an example where the finger of a robot is controlled with two tasks simultaneously. Those two tasks have different target locations thus they conflict with each other. To understand the behavior of the inverse kinematics in this situation the loss function $\mathcal{L}$ from equation (3.5) has to be recalled:

$$\mathcal{L}(\Delta q) = ||y^* - \psi(q + \Delta q)||_C^2 + ||\Delta q||_W^2$$

The term $||y^* - \psi(q + \Delta q)||_C^2$ assures that the end effector is moved to a state where the overall error is minimized. In the given example the state minimizing $\mathcal{L}$ is the location right between the two targets.



(a) Multiple tasks for a two-armed robot

(b) Conflicting tasks for a single effector

Figure 4.2: Solving multiple tasks simultaneously

In Figure 4.2 the initial posture is displayed dashed. Since the solution process for (dampened) *Newton-Raphson* approaches needs several iterations to reach the posture minimizing $\mathcal{L}$ each iteration is displayed in grayscale; the higher the iteration number the darker the posture. For each iteration the vector from the respective end effector to the task's target is drawn.

The algorithm implementing inverse kinematics with combined tasks is:

Listing 4.3: Inverse kinematics algorithm with augmented Jacobians and error vectors

```
def solve_ik(robot, taskGroup, epsilon):
  J_s   = buildStackedJacobian(robot, taskGroup)
  error = buildStackedError(robot, taskGroup)
  dq  = pinv(J_s, epsilon) * error
  return dq
```

5 - The Stack of Tasks

Given a hierarchy of tasks:
How can tasks with a lower priority be solved without interfering with tasks of higher priority?

The previous chapter has shown how multiple tasks can be solved simultaneously. This is useful when dealing with complex robotic systems but does not allow expressing a hierarchy of tasks. A hierarchy might be an ordered set of sets of tasks with the constraint:

$$0 = \hat{J}_{1,i-1} \Delta q_i \tag{5.1}$$

This means the change in configuration which is calculated for the i-th group of tasks must *not* create a change in task space of all task groups with higher priority. Mansard et al. refer to this hierarchy as *stack of tasks*[17]. The total Δq then can be calculated as:

$$\boxed{\Delta q = \sum_i \Delta q_i} \tag{5.2}$$

Equation (5.1) can also be formulated with the rangespace projector of J_i:

$$0 = {J_{1,i-1}}^{\dagger} J_{1,i-1} \Delta q_i \tag{5.3}$$

For $J_{1,i-1}$ having redundant effectors:

$${J_{1,i-1}}^{\dagger} J_{1,i-1} \neq I \tag{5.4}$$

There are nontrivial solutions to (5.1) with $\Delta q_i \neq 0$. Namely all Δq that reside inside the combined nullspace of all $J_{1,i-1}$. An intuitive solution to find Δq_is that satisfy (5.1) was proposed by [6, 8–10, 12] and [14] [1]:

$$\Delta q_j = \mathcal{N}_i \underbrace{{J_{1,i-1}}^{\dagger} \vec{e}_j}_{\Delta \hat{q}} \tag{5.5}$$

$\mathcal{N}_i$ projects the change in configuration space – when only considering the task group i $\Delta \hat{q}$ – into the nullspace of $J_{1,i-1}$. This satisfies the criterion (5.1):

[1] Chiaverino et al. and Bock et al. recognized this approach to be non-optimal.

$$
\begin{aligned}
0 &= J_i \Delta q_j \\
0 &= J_i \mathcal{N}_i {J_j}^\dagger \vec{e}_j \\
0 &= J_i \left(I - {J_i}^\dagger J_i \right) {J_j}^\dagger \vec{e}_j \\
0 &= \left(J_i - J_i {J_i}^\dagger J_i \right) {J_j}^\dagger \vec{e}_j
\end{aligned}
$$

by applying (3.14)

$$
\begin{aligned}
0 &= \left(J_i - J_i \right) {J_j}^\dagger \vec{e}_j \\
0 &= 0 {J_j}^\dagger \vec{e}_j
\end{aligned}
$$

□

However, (5.5) does not provide good solutions to solve the task group i as $\mathcal{N}_{1,i-1}$ can scale $\Delta\hat{q}$ in a way that the (5.5) does not yield the optimal Δq_j with respect to the loss function $\mathcal{L}$. To visualize the non-optimality of the proposed solution a robot arm was simulated and the results are shown in figure 5.1. The arm had to fulfill two tasks:

- First task:
  Move the finger onto a horizontal line with support point $\begin{pmatrix} 1 & 0 \end{pmatrix}^T$ (the dotted line). This line is the solution space of the first task.
- Second task:
  Without violating the first task: move the finger to the location denoted with the dashed vectors.

The second task is solved within the nullspace of the first as described in (5.5).

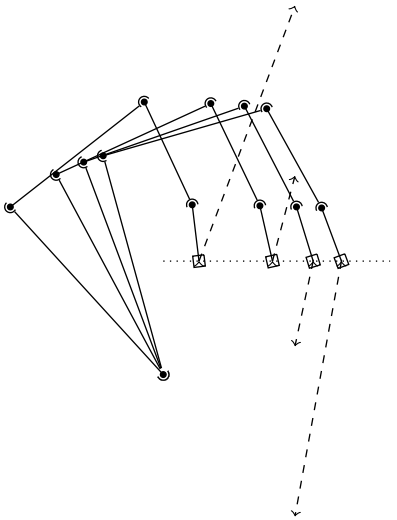


Figure 5.1: Non-optimal utilization of motion within the nullspace of a task with higher priority

The optimal solutions for the second task would be at the perpendicular projection of the targets onto the solution space of the first task. But the figure shows that the further away the target of the second task is from the line the further the finger moves away from the optimal solution. This example is sufficient proof that (5.5) is not optimal.

To find out how to utilize $\mathcal{N}$ in an optimal fashion and while fulfilling the constraint (5.1) one has to revisit the loss function (3.5) from section 3.3:

$$\mathcal{L}(\Delta q) = ||y^* - \psi(q + \Delta q)||_C^2 + ||\Delta q||_W^2$$

W is the matrix that "punishes" changes for joints. The greater the entries in W the greater the "punishment". W can also be used to express changes of joints that have to happen in conjunction as a linear combination. To integrate constraints into the loss function W has to be set to $\mathcal{N}^{-1}$ as it punishes usage of joints that are already in use by other tasks. This is advantageous as $\mathcal{N}$ has the following properties:

- $\mathcal{N}$ is Hermitian: $\mathcal{N} = \mathcal{N}^T$.
- The eigenvalues of $\mathcal{N}$ are either 0 or 1 $\Rightarrow$ eigenvalues of $\mathcal{N}^{-1}$ would be either ∞ or 1.
- The elements of each row encode joint movements that have to be performed in conjunction (the same applies to the columns as $\mathcal{N}$ is Hermitian). Here the i-th row (and therefore the i-th column) work as a linear combination in which the joints' values have to be changed to not violate the tasks that generate $\mathcal{N}$.
- The elements along the main diagonal are in the interval $[0, 1]$.

With those properties an entry of 0 along the main diagonal of $\mathcal{N}$ implies that the corresponding row and column is filled with zeros as well. This means that any joint i for which $\mathcal{N}_{i,i} = 0$ is completely bound by other tasks and its value cannot be changed without creating a violation of tasks with higher priority. Whereas $\mathcal{N}_{i,i} \neq 0$ means that the joint i can be fully utilized ($\mathcal{N}_{i,i} = 1$) or only be utilized in conjunction with other joints ($0 < \mathcal{N}_{i,i} < 1$). However, $\mathcal{N}$ is generally not invertible but by integrating $\mathcal{N}$ into equation (3.9) it becomes clear that building the inverse of $\mathcal{N}$ is not necessary.

$$\Delta q = W^{T^{-1}} J^T (J W^{T^{-1}} J^T + C^{T^{-1}})^{-1} \vec{e}$$

With $W^T = \mathcal{N}^{-1}$:

$$\boxed{\Delta q = \mathcal{N} J^T (J \mathcal{N} J^T + C^{T^{-1}})^{-1} \vec{e}} \tag{5.6}$$

$\mathcal{N} J^T (J \mathcal{N} J^T + C^{T^{-1}})^{-1}$ is *not* the pseudoinverse of J but the pseudoinverse of $J\mathcal{N}$:

$$\mathcal{N} J^T (J \mathcal{N} J^T + C^{T^{-1}})^{-1}$$

$\mathcal{N}$ is idempotent: $\mathcal{N}\mathcal{N} = \mathcal{N}$

$$\begin{aligned}
&= \mathcal{N}^T J^T (J \mathcal{N} \mathcal{N} J^T + C^{T^{-1}})^{-1} \\
&= \mathcal{N}^T J^T (J \mathcal{N} \mathcal{N}^T J^T + C^{T^{-1}})^{-1} \\
&= (J\mathcal{N})^T ((J\mathcal{N})(J\mathcal{N})^T + C^{T^{-1}})^{-1}
\end{aligned}$$

with $\lim_{C \to \infty}$:

$$= (J\mathcal{N})^\dagger \tag{5.7}$$

Equation (5.7) can also be interpreted as a single task that provides the Jacobian $\bar{J} = J\mathcal{N}$. In terms of the loss function (3.5) this would be the solution to having W set to I thus there would be no punishment for the usage of some joints. This does not conflict with (3.5) as the usability of joints is already encoded in $\bar{J}$ [2]. Murray et al. have shown that the integration of $\mathcal{N}$ into the least-squares solution process is the same as solving tasks of lesser priority as a constrained optimization problem using the *Lagrangian relaxation* method [18].

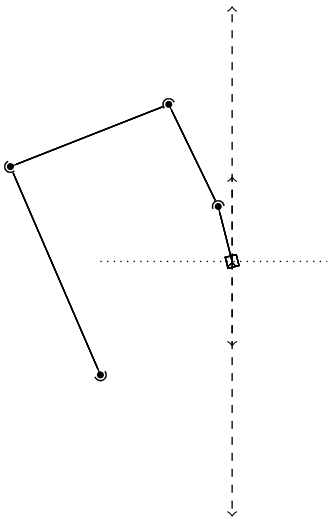


Figure 5.2: Optimal utilization of motion within the nullspace of a task with higher priority

Figure 5.2 shows the generated postures of an inverse kinematics solver implementing (5.6). In terms of the reduced solution space of the second task the generated locations of the finger are in fact optimal: The second task's target projected orthogonally onto the first tasks solution space.

5.1 Improving the Numerical Stability

It is worth noting that the $\mathcal{N}$ utilized above is the *dampened* nullspace projector (as discussed in section 3.4) and thus an approximation of the actual nullspace projector. This approximation is subject to numerical inaccuracies which results in $\text{trace}(\mathcal{N}) \notin \mathbb{N}$ (see section 3.4). Especially for $\mathcal{N} \to 0$ this can lead to numerical instabilities. A mitigation strategy is to multiply $\mathcal{N}^2$ to the left hand side of $(J\mathcal{N})^\dagger$. The resultant pseudoinverse becomes much more stable and utilizable while retaining the pseudoinverse properties. To emphasize the reason why this step is preferable the rule of *L'Hôpital* can be applied to show the behavior of $(J\mathcal{N})^\dagger$ for $\mathcal{N} \to 0$. The desired behavior is:

$$\lim_{\mathcal{N}\to 0} (J\mathcal{N})^\dagger = 0$$

but:

$$\lim_{\mathcal{N}\to 0} \lim_{\epsilon\to 0} \mathcal{N}J^T(J\mathcal{N}\mathcal{N}J^T + \epsilon I)^{-1} \neq 0$$

[2] This also means that the workspace of the task with lesser priority is limited by $\mathcal{N}$ which renders investigation of workspaces cumbersome – especially when the Jacobian that generates $\mathcal{N}$ is ill-conditioned [14].

because by applying the rule of *L'Hôpital*:

$$\lim_{\mathcal{N}\to 0}\lim_{\epsilon\to 0}\mathcal{N}J^T(J\mathcal{N}\mathcal{N}J^T+\epsilon I)^{-1} \Rightarrow J*(2J^T\mathcal{N}J)^{-1}=\infty \tag{5.8}$$

However, by expanding the leftmost $\mathcal{N}$ to $\mathcal{N}^3$:

$$\lim_{\mathcal{N}\to 0}\lim_{\epsilon\to 0}\mathcal{N}^3J^T(J\mathcal{N}\mathcal{N}J^T+\epsilon I)^{-1} \Rightarrow 3J\mathcal{N}^2*(2J^T\mathcal{N}J)^{-1} \Rightarrow 4\mathcal{N}J^T*(2JJ^T)^{-1}=0 \tag{5.9}$$

With this addition the numerical stability can be greatly improved. And since $\mathcal{N}$ is idempotent this approach is optimal with respect to the loss function $\mathcal{L}$ introduced in section 3.3.

5.2 Combining Nullspaces

Finding the combined nullspace projector of several matrices is necessary when the stack of tasks becomes big. The combined nullspace projector is a matrix that projects any vector into all nullspaces spanned by all Jacobians of more prioritized tasks. When solving for the i-th group of tasks all task groups j with $j<i$ shall not be violated. This combined nullspace projector has to fulfill:

$\forall j<i, \vec{p}$:

$$0=J_j\mathcal{N}_i\vec{p}$$

The trivial way to achieve this is to calculate $\mathcal{N}_i$ by using the augmented Jacobian of all $j<i$:

$$\hat{J}=\begin{pmatrix}J_1\\ \vdots \\ J_j\end{pmatrix}$$
$$\mathcal{N}_i=I-\hat{J}^\dagger\hat{J} \tag{5.10}$$

Mansard et al. proposed a recursive algorithm to calculate $\mathcal{N}_i$ that is computationally significantly faster[30]:

$$\mathcal{N}_i=\mathcal{N}_{i-1}-\underbrace{(J_i\mathcal{N}_{i-1})^\dagger J_i\mathcal{N}_{i-1}}_{\mathcal{M}_i} \tag{5.11}$$

The above algorithm works by subtracting the range projector of the i-th group of tasks from the available solution space for each iteration. This successively reduces the joint solution space – which is the nullspace. However, it is *not* numerically stable when any J is ill-conditioned as $J_i\mathcal{N}_{i-1}$ might contain base vectors of minuscule lengths or when i becomes big. The core problem in (5.11) is the subtraction of the range projector of $J_i\mathcal{N}_{i-1}$ from $\mathcal{N}_{i-1}$. Any numerical error introduced in the calculation of $\mathcal{M}_i$ will introduce more numerical errors into nullspace projector in which tasks of lower priorities are solved in. More generally, when working with big stacks of tasks this algorithm – and any other algorithm that works accumulatively – will not perform well. Even adjusting (5.11) as proposed in (5.9) the numerical stability cannot be guaranteed as the algorithm would still rely on an accumulated difference. The numerically stable calculation of $\mathcal{N}_i$ is described in (5.10).
A simple implementation of the inverse kinematics solver proposed with (5.6) is:

Listing 5.1: Simple inverse kinematics algorithm incorporating the stack of tasks

```
def solve_ik(robot, taskStack, epsilon, nullspaceEpsilon):
  numCols = robot.getDOF()
  I = np.matrix(np.eye(numCols, numCols))
  q = robot.getConfiguration()
  J_c = np.matrix(np.zeros(0, numCols))
  for i in range(0, len(taskStack)):
    Ny = I - pinv(J_c, nullspaceEpsilon) * J_c
    jacobian = buildStackedJacobian(robot, [taskStack[i]])
    error    = buildStackedError(robot, taskStack[i])
    dq = Ny * Ny * pinv(jacobian * Ny, epsilon) * error
    q += dq
    J_c = np.concatenate((J_c, jacobian))

def buildStackedJacobian(robot, taskGroups):
  j = np.matrix(np.zeros((0, robot.getDOF())))
  for taskGroup in taskGroups:
    for task in taskGroup:
      j = np.concatenate((j, task.getJacobian()))
  return j

def buildStackedError(robot, taskGroup):
  e = np.matrix(np.zeros((0, 1))
  for task in taskGroup:
    e = np.concatenate((e, task.getError()))
  return e
```

$\mathcal{N}$ is derived from `J_c` while in each loop `J_c` is augmented with the current loop's Jacobian. Each Jacobian that is calculated within the loop is solely dependent on the initial posture of the robot. However, while iterating over the stack of tasks the algorithm produces intermediate changes in the robot's posture that will be performed anyway. That is when calculating Δq_i the value of $\sum_{j}^{j<j} \Delta q_j$ is already known but neither utilized in the generation of the i-th Jacobian nor in the nullspace in which the i-th group has to be solved in. While this algorithm is valid it is not optimal and can be improved by recalculating all Jacobians that are utilized within each iteration of the loop. A better implementation is depicted in listing 5.2. Here the robot's posture is adjusted each time a Δq_i is calculated. Because Jacobians are dependent on q they have to be recalculated for the next group of tasks. This process yields better results and faster rate of convergence.

Listing 5.2: Improved inverse kinematics algorithm

```
def solve_ik(robot, taskStack, epsilon, nullspaceEpsilon):
  numCols = robot.getDOF()
  I = np.matrix(np.eye(numCols, numCols))
  q = robot.getConfiguration()
  for i in range(0, len(taskStack)):
    J_c = buildStackedJacobian(robot, taskStack[0:i])
    Ny = I - pinv(J_c, nullspaceEpsilon) * J_c
    jacobian = buildStackedJacobian(robot, [taskStack[i]])
    error    = buildStackedError(robot, taskStack[i])
    q += Ny * Ny * pinv(jacobian * Ny, epsilon) * error
    robot.setConfiguration(q)
```

6 - Loopy Robots - Linear Constraints

How can inverse kinematics be applied to robots with nontrivial structures?

The previous chapters of this thesis rely on robots having a treelike structure. However, in real world applications this is not always the case. When dealing with robots that have multiple arms and need to interact with an object using both hands the robot becomes *loopy* as soon as the hands hold on the same object or each other [31]. Another case for loopy robots are humanoid robots that stand on the ground with both feet. Here the ground connects both feet. Even though the feet could slide on the floor it should be avoided. As long as both feet connect to the ground the desired behavior is to have both feet at a fixed position and rotation with respect to each other [30].

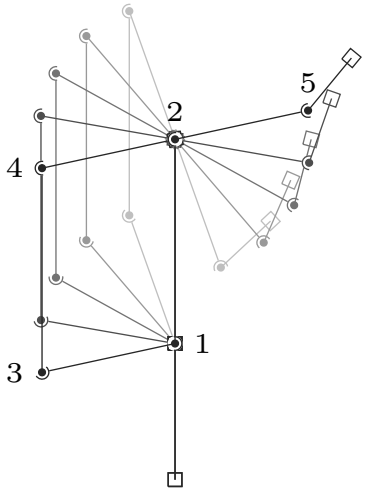


Figure 6.1: A loopy robot with rhomboid enforced structure

Figure 6.1 shows a robot with five revolute joints where four of its joints enforce a structure that resembles a rhomboid. This robot can also be described with a loop-free topology (see figure 6.2) and additional constraints that keep the node $2'$ congruent and oriented parallel to node 2. Those constraints are expressed as a location task that keeps the location of node $2'$ at position $\begin{pmatrix}0 & 0\end{pmatrix}^T$ as seen from node 2 in combination with an orientation task that fixes the orientations of both nodes as well. With those constraints any other task that introduces movement in the robots structure is solved within the nullspace of both constraining tasks.

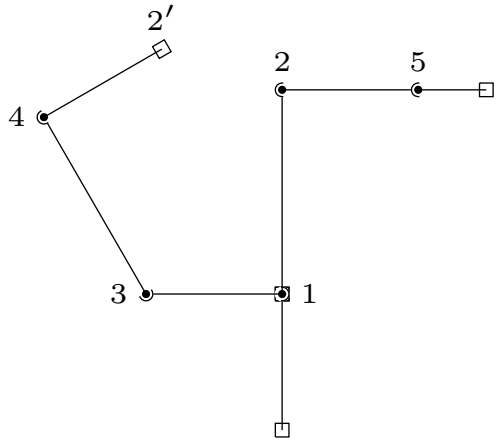


Figure 6.2: Tree structure of a loopy robot

Figure 6.2 shows the same robot as in figure 6.1 without the constraints to emphasize the robot's basic tree structure. The ability to build loopy robotic systems also enables the construction of nontrivial joints; structures consisting of multiple revolute and/or prismatic joints working in conjunction thus forming a single more complex joint. The rhomboid-like joint in figure 6.1 is such a case. When building a robot with that kind of structure a single motor can be attached anywhere on one of the joints 1 to 4 controlling the whole rhomboid structure simultaneously. To find out how many degrees of control are needed to fully control a loopy subsystem trace of the range projector of the combined task for the constraints $\mathcal{M}_c$ can be used. In the given example the range projector is:

$$\mathcal{M}_c = J_c^\dagger J_c$$

$$= \begin{pmatrix} \overbrace{\begin{matrix} 0.75 & 0.25 & -0.25 & -0.25 \\ 0.25 & 0.75 & 0.25 & 0.25 \\ -0.25 & 0.25 & 0.75 & -0.25 \\ -0.25 & 0.25 & -0.25 & 0.75 \end{matrix}}^{*1} & \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 & 0 \end{matrix} & \underbrace{0}_{*2} \end{pmatrix}$$

- *1: This submatrix reflects that joints 1 to 4 have to be changed as a linear combination to not violate the constraints.
- *2: This column corresponds to the joint 5 that is not part of the loopy subsystem.

As the loop in this example consists of four joints and $\text{trace}(\mathcal{M}_c) = 3$ there is a rank defect of 1. This means that the loopy part of the robot's structure resembles a single degree of freedom and therefore can be completely controlled by attaching a single motor on any joint on the loop. Also, because the constraints for the loopy system can be fully described with a matrix – a linear transformation – for any q those constraints are *linear constraints*.

7 - Nonlinear Constraints

How can joint limits be reflected in inverse kinematics?

Nonlinear constraints – or *unilateral constraints* – are derived from nonlinear functions e.g., limits of joint values. When the inverse kinematics finds a solution to adjust a posture the solution might violate those constraints. That happens for instance when an arm is bent in a way it is not supposed to be bent or when a piston joint is stretched beyond its limits.

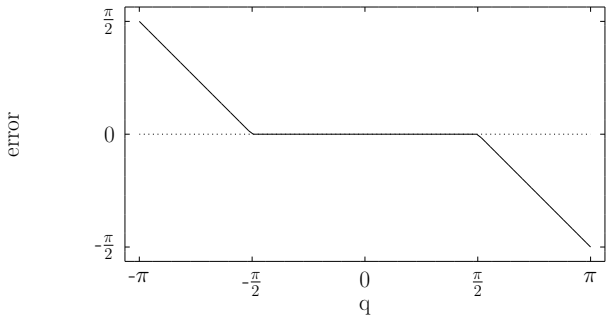


Figure 7.1: Error function for nonlinear constraints

Figure 7.1 displays an exemplary error function for a joint where the valid values are in the interval $\left[-\frac{pi}{2}, \frac{pi}{2}\right]$. The function's output is the change that has to be performed on that joint to move the joint's value into the correct interval. This function is *continuous* but *not continuously differentiable*. However, joint limits are very common and should be taken care of when solving for inverse kinematics because movements beyond those limits might damage the robot.
To integrate unilateral constraints into the solution process of inverse kinematics a function has to be implemented that returns a vector q_e containing the error for each joint. This function resembles the function displayed in 7.1 but for each element in q and with different parameters for each joint. Mansard et al. [17] proposed a technique to integrate an *activation matrix* $H \in \mathbb{R}^{n \times n}$:

$$H = \text{diag}(h_1, h_2, \ldots, h_n)$$

where:

$$h_i = \begin{cases} 1, & \text{if } q_e \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{7.1}$$

It is worth emphasizing that their approach integrates unilateral constraints into the *control law* derived from inverse kinematics. This is not the same as integration into general inverse kinematics as *control* is usually expressed in forces (torques) or speeds but not values (e.g., angles). But the approach proposed by Mansard et al. can be used as a starting point to build a concept of how unilateral constraints can be integrated into general inverse kinematics.
The above introduced activation matrix H is then utilized to build a modified Jacobian for any task that should be solved for:

$$J^{\dagger^H} = \sum_{P \in \mathfrak{P}(m)} (\prod_{i \in P} h_i) \prod_{j \notin P} (1 - h_i) J^{\dagger}{}_P \tag{7.2}$$

In equation (7.2) $\mathfrak{P}(m)$ denotes the power set of the set of all integers ranging from 1 to m. m is the amount of degrees of freedom of the robot. The idea behind this equation is to suppress degrees of freedom if they are subject to errors in configuration space. This happens when joint limits are violated. With this the control output is calculated as:

$$\dot{q} = J^{\dagger^H} * \dot{e} \tag{7.3}$$

In terms of control law this approach is valid as it is applied continuously and there exist no time discrete steps where the control process coerces a joint to move beyond its boundaries of valid configurations. However, to build an inverse kinematics solver that can solve for postures the process of finding a valid solution is *not* continuous. Also, for robotic systems with many degrees of freedom the equation (7.2) results in computationally unfeasible performance.
Another approach to resolve joint limits was proposed by Chan et al. [8] and Dietrich et al. [29]. They implement a scheme to avoid joint limits by incorporating a weighted least-norm solver. The idea is to dampen joint's movements when they are near their respective limits and reflect this within a matrix that is similarly incorporated as $\mathcal{N}$ in the previous chapters. The drawback in their approach is that by dampening joints near their respective limits the actual workspace of robots is limited or at least the rate of convergence is reduced.
Generally there is no trivial way to integrate nonlinear constraints into least-squares inverse kinematics because the foundation of the later resides on the linearity of the local derivative. But when any q_i is at the corresponding joint limit the local tangent would be different depending on the change that is applied on q_i by further iterations. If the lower limit of joint i is set to 0 and the current value is 0 then $\Delta q_i >= 0$ would be valid but a change where $\Delta q_i < 0$ needs to be avoided in any event. This cannot be expressed using matrices due to their linear nature.
Therefore, each Δq has to be tested whether the resultant posture $q_{t+1} = \Delta q + q_t$ violates any constraint or not. Given the assumption the robotic system is not violating any constraint when being at posture q_t but q_{t+1} violates one or more constraints the error was introduced by Δq. Hence, Δq has to be adjusted to resolve the error. This leads to q_{t+q} being free of errors. A simple way to resolve this would be to *backtrack* the error introduced with Δq. To achieve this the error can be transformed into an error in task space – this can be done since the error was introduced by solving for a task in the first place:

$$e_t = J * q_e \tag{7.4}$$

e_t then can be reprojected into configuration space along the solution direction of the current task by applying:

$$\Delta \hat{q}_e = J^{\dagger} J * q_e \tag{7.5}$$

Here $\Delta\hat{q}$ is the change that has to be applied to Δq to resolve the constraints. But due to the nonlinear nature of the error function the resultant $q_t + \Delta q + \Delta\hat{q}_e$ does not necessarily resolve the issue of being free of errors or at the configuration where the joint limits are just reached. In addition, it does not guarantee that by backtracking along the direction of $J^\dagger J q_e$ the resultant task error is less than the initial task error at q_t. A better approach is to iteratively backtrack for each introduced q_{e_i} starting at the greatest entry of q_e along the direction of Δq until all errors introduced by constraints are resolved. After the backtracking the last joint for which the backtracking had to be performed is "fixed" and the process of solving the task is repeated using the remaining free degrees of freedom by incorporating a modified $\mathcal{N}$ matrix that reflects fixed joints.

Listing 7.1: Backtracking algorithm

```
def backtrack(dq, q)
  idx = -1
  q += dq
  q_e = calculateQError(q)
  while (abs(q_e) > 0).any():
    idx = abs(q_e).argmax()
    f = dq_[idx,0] / q_err[idx, 0]
    q_corrected = dq * 1 / f
    q += q_corrected
    q_e = calculateQError(q)
  return idx
```

Algorithm 7.1 points out how to perform the backtracking. Note that the arguments `dq` and `q` are passed as references. Within the while-loop Δq is backtracked along the greatest entry of q_e which is then repeated for all remaining errors. Under the assumption that the error function for each joint is built as depicted in figure 7.1 the last iteration of the loop guarantees that the joint for which the last backtacking step was executed is right at its limit. Furthermore, the configuration vector q_{i+q} (denoted in the code as `q`) is changed in a way that the error is resolved.
The integration in the inverse kinematics solving algorithm then is:

Listing 7.2: Inverse kinematics solver incorporating the stack of tasks and nonlinear constraints

```
def solve_ik(robot, taskStack, epsilon, nullspaceEpsilon):
  numDOF = robot.getDOF()
  dq = np.matrix(np.zeros((numDOF, 1)))
  q = robot.getConfiguration()
  I = np.matrix(np.eye(numDOF, numDOF))
  for i in range(len(taskStack)):
    extraConstraints = np.matrix(np.zeros((numDOF, numDOF)))
    while True:
      if np.sum(np.diag(extraConstraints, 0)) == numDOF:
        break
      robot.setConfiguration(q)
      J_c = buildStackedJacobian(robot, tasks[0:i])
      J_c = np.concatenate((J_c, extraConstraints))
      Ny = I - pinv(J_c, nullspaceEpsilon) * J_c
      jacobian = buildStackedJacobian(robot, [taskStack[i]])
      error    = buildStackedError(robot, taskStack[i])
      dq = Ny * Ny * pinv(jacobian * Ny, epsilon) * error
```

```
18        q += dq
19        q_e = calculateQError(q)
20        if (abs(q_err) > 0).any():
21          idx = backtrack(q, dq)
22          extraConstraints[idx, idx] = 1
23        else:
24          break
25      robot.setConfiguration(q)
26    return q
```

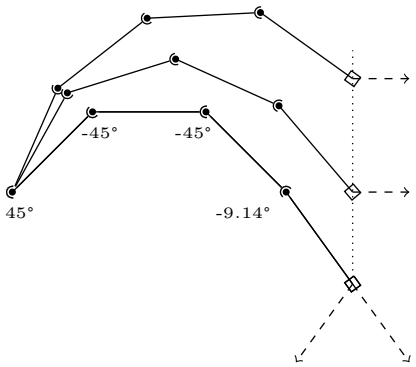


Figure 7.2: Unilateral constraints

Figure 7.2 is generated by the algorithm shown in listing 7.2. Here the robot arm performs two tasks while respecting the joint limits on each joint:

- Move the finger onto the dotted line.
- Without violating the first task move the finger to the position denoted with the dashed arrows.

The joint limits are set to the interval $[-45°, 45°]$ except for the first joint where it is $[45°, 90°]$. Figure 7.2 shows that when the target of the second task is set out of reach the position of the robot's finger rests at the nearest configuration (displayed as the bottommost posture) to the target without bending the joints beyond the respective limits.

8 - Conclusion and Future Work

The contribution of this work is the conclusive derivation of the dampened least-squares approach to inverse kinematics. This derivation is based on a loss function which also acts as a quality measure as well as means to prove extensions for optimality. Furthermore, an extension to improve numerical stability, an extension to handle loopy robots and an approach to handle nonlinear constraints are introduced with this work.

Forward kinematics lay the foundation to compute the *Jacobian* matrices and error vectors for different task types as shown in chapters 2 to 4. With this the ability to calculate static and dynamic properties of arbitrary robots is implemented. Chapter 3 is the main part of this work where the *Newton-Raphson* gradient descent is outlined. This gradient descent technique is then utilized to minimize the square *loss function* $\mathcal{L}$. $\mathcal{L}$ acts as metric that is used in the later chapter to test extensions of inverse kinematics against optimality. This chapter also shows that the dampened *Moore-Penrose pseudoinverses* of Jacobian matrices are the key to solve the inverse kinematics problem. How hierarchies of tasks can be expressed is shown in chapter 5. This hierarchy can be utilized to extend inverse kinematics to nontrivial robot structures e.g., *loopy robots*. Lastly, to avoid movements that cannot be performed on joint level the framework can be extended to incorporate *joint-limit* constraints.

Future Work

While the dampened least-squares approach to inverse kinematics is very versatile there are still properties that can be improved: When a task's target is outside its workspace the gradient descent will not converge stably (see 3.3). Also, the choice of the dampening factor ϵ is cumbersome. A high dampening appears to counter the instability of convergence when a task's target is out of reach but increasing the dampening mitigates this only up to a certain extend. In addition, the overall rate of convergence is highly impacted by the dampening factor (as shown in figure 3.5). This work focuses rigid robotic systems. Real world robots might contain springs or other deformable parts that cannot be modeled with the means introduced in this thesis.

Bibliography

[1] Roy Featherstone and David Orin. "Chapter 2: Dynamics". In: *Springer Handbook of Robotics* (2008) (cit. on pp. 2, 3).

[2] David Orin and William W Schrader. "Efficient computation of the Jacobian for robot manipulators". In: *The International Journal of Robotics Research* 3.4 (1984), pp. 66–75 (cit. on pp. 4, 24).

[3] C. W. Wampler. "Manipulator Inverse Kinematic Solutions Based on Vector Formulations and Damped Least-Squares Methods". In: *IEEE Transactions on Systems, Man and Cybernetics* 16.1 (Jan. 1986), pp. 93–101. ISSN: 0018-9472. DOI: 10.1109/TSMC.1986.289285 (cit. on pp. 8, 11, 14).

[4] K Waldron et al. "Springer handbook of robotics". In: *Springer, Berlin, Heidelberg, New York* (2008) (cit. on p. 10).

[5] Donald Lee Pieper. *The kinematics of manipulators under computer control*. Tech. rep. DTIC Document, 1968 (cit. on p. 11).

[6] G. Antonelli. "Stability Analysis for Prioritized Closed-Loop Inverse Kinematic Algorithms for Redundant Robotic Systems". In: *IEEE Transactions on Robotics* 25.5 (Oct. 2009), pp. 985–994. ISSN: 1552-3098. DOI: 10.1109/TRO.2009.2017135 (cit. on pp. 14, 32).

[7] Layale Saab et al. "Dynamic whole-body motion generation under rigid contacts and other unilateral constraints". In: *Robotics, IEEE Transactions on* 29.2 (2013), pp. 346–362 (cit. on p. 14).

[8] Tan Fung Chan and Rajiv V Dubey. "A weighted least-norm solution based scheme for avoiding joint limits for redundant joint manipulators". In: *Robotics and Automation, IEEE transactions on* 11.2 (1995), pp. 286–292 (cit. on pp. 14, 32, 41).

[9] Oliver Brock, Oussama Khatib, and Sriram Viji. "Task-consistent obstacle avoidance and motion behavior for mobile manipulation". In: *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*. Vol. 1. IEEE. 2002, pp. 388–393 (cit. on pp. 14, 32).

[10] Joel W Burdick. "On the inverse kinematics of redundant manipulators: Characterization of the self-motion manifolds". In: *Advanced Robotics: 1989*. Springer, 1989, pp. 25–34 (cit. on pp. 14, 32).

[11] Samuel R Buss and Jin-Su Kim. "Selectively damped least squares for inverse kinematics". In: *journal of graphics, gpu, and game tools* 10.3 (2005), pp. 37–49 (cit. on pp. 14, 18).

[12] Yu-Che Chen and Ian D Walker. "A consistent null-space based approach to inverse kinematics of redundant robots". In: *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*. IEEE. 1993, pp. 374–381 (cit. on pp. 14, 32).

[13] Stefano Chiaverini. "Singularity-robust task-priority redundancy resolution for real-time kinematic control of robot manipulators". In: *Robotics and Automation, IEEE Transactions on* 13.3 (1997), pp. 398–410 (cit. on pp. 14, 19).

[14] Stefano Chiaverini, Giuseppe Oriolo, and Ian D Walker. "Kinematically Redundant Manipulators". In: *Springer Handbook of Robotics*. Springer, Jan. 1, 2008. ISBN: 9783540239574. DOI: 10.1007/978-3-540-30301-5_12 (cit. on pp. 14, 30, 32, 35).

[15] Wankyun Chung, Li-Chen Fu, and Su-Hau Hsu. "Motion Control". In: *Springer Handbook of Robotics*. Springer, Jan. 1, 2008. ISBN: 9783540239574. DOI: 10.1007/978-3-540-30301-5_7 (cit. on p. 14).

[16] Oussama Khatib et al. "Whole-body dynamic behavior and control of human-like robots". In: *International Journal of Humanoid Robotics* 1.01 (2004), pp. 29–43 (cit. on p. 14).

[17] Nicolas Mansard, Oussama Khatib, and Abderrahmane Kheddar. "A Unified Approach to Integrate Unilateral Constraints in the Stack of Tasks". In: *IEEE Transactions on Robotics* 25.3 (June 2009), pp. 670–685. ISSN: 1552-3098. DOI: 10.1109/TRO.2009.2020345 (cit. on pp. 14, 30, 32, 40).

[18] Richard M Murray et al. *A mathematical introduction to robotic manipulation*. CRC press, 1994 (cit. on pp. 14, 15, 18, 35).

[19] Hamid Sadeghian et al. "Task-space control of robot manipulators with null-space compliance". In: *Robotics, IEEE Transactions on* 30.2 (2014), pp. 493–506 (cit. on p. 14).

[20] Luis Sentis and Oussama Khatib. "Prioritized multi-objective dynamics and control of robots in human environments." In: *Humanoids*. 2004, pp. 764–780 (cit. on p. 14).

[21] B. Siciliano and J.-J. E. 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*. June 1991, 1211–1216 vol.2. DOI: 10.1109/ICAR.1991.240390 (cit. on p. 14).

[22] D. E. Whitney. "Resolved Motion Rate Control of Manipulators and Human Prostheses". In: *IEEE Transactions on Man-Machine Systems* 10.2 (June 1969), pp. 47–53. ISSN: 0536-1540. DOI: 10.1109/TMMS.1969.299896 (cit. on p. 14).

[23] Marc Toussaint. *Robotics Lecture 2013*. online. Accessed 2015-03-28. URL: https://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/13-Robotics/02-kinematics.pdf (cit. on pp. 14, 16).

[24] Michael Gienger, Marc Toussaint, and Christian Goerick. "Task maps in humanoid robot manipulation". In: *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE. 2008, pp. 2758–2764 (cit. on p. 15).

[25] Max A Woodbury. "Inverting modified matrices". In: *Memorandum report* 42 (1950), p. 106 (cit. on p. 16).

[26] Roger Penrose. "A generalized inverse for matrices". In: *Proc. Cambridge Philos. Soc.* Vol. 51. 3. Cambridge Univ Press. 1955, pp. 406–413 (cit. on p. 17).

[27] Gene Golub and William Kahan. "Calculating the singular values and pseudo-inverse of a matrix". In: *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis* 2.2 (1965), pp. 205–224 (cit. on p. 18).

[28] Lars Eldén. "A weighted pseudoinverse, generalized singular values, and constrained least squares problems". In: *BIT Numerical Mathematics* 22.4 (1982), pp. 487–502 (cit. on p. 18).

[29] A. Dietrich, A. Albu-Schaffer, and G. Hirzinger. "On continuous null space projections for torque-based, hierarchical, multi-objective manipulation". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. May 2012, pp. 2978–2985. DOI: 10.1109/ICRA.2012.6224571 (cit. on pp. 18, 41).

[30] Nicolas Mansard et al. "A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The stack of tasks". In: *Advanced Robotics, 2009. ICAR 2009. International Conference on*. IEEE. 2009, pp. 1–6 (cit. on pp. 30, 36, 38).

[31] A. M. Zanchettin and P. Rocco. "Dual-arm redundancy resolution based on null-space dynamically-scaled posture optimization". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. May 2012, pp. 311–316. DOI: 10.1109/ICRA.2012.6224599 (cit. on p. 38).