

# Masterarbeit

Titel der Arbeit // Title of Thesis

**Erweiterung der Roboterprogrammierung um die  
Kollisionserkennung und Konfigurationsbestimmung,  
durch Simulation und Analyse der Kinematik**

Extension of Robot Programming by Collision Detection and Configuration  
Determination, by Simulation and Analysis of Kinematics

---

Akademischer Abschlussgrad: Grad, Fachrichtung (Abkürzung) // Degree  
**Master of Engineering (M.Eng.)**

---

Autorenname, Geburtsort // Name, Place of Birth  
**René Horstmann, Bocholt**

---

Studiengang // Course of Study  
**Master Maschinenbau Fachrichtung Robotik**

---

Fachbereich // Department  
**Maschinenbau**

---

Erstprüferin/Erstprüfer // First Examiner  
**Prof. Dr.-Ing. Dr. h.c. Dieter Schramm**

---

Zweitprüferin/Zweitprüfer // Second Examiner  
**Prof. Dr. Antonio Nisch**

---

Abgabedatum // Date of Submission

**15.08.2018**

---



## **Eidesstattliche Versicherung**

**Horstmann, René**

---

Name, Vorname // Name, First Name

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem Titel  
**Erweiterung der Roboterprogrammierung um die  
Kollisionserkennung und Konfigurationsbestimmung, durch  
Simulation und Analyse der Kinematik**

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen  
als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße  
Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner  
Prüfungsbehörde vorgelegen.

---

Ort, Datum, Unterschrift // Place, Date, Signature

---

## Zusammenfassung

In dieser Arbeit wird eine Roboterbibliothek mit Kollisionserkennung und Konfigurationsbestimmung entwickelt. Zielsetzung ist, einen Industrieroboter mithilfe von zuvor definierten Prozessbahnen kollisionsfrei im Raum zu manövrieren. Hierzu wird dessen Kinematik analysiert, um somit mögliche Konfigurationen zum Erreichen einer Pose zu ermitteln. Diese werden in anschließender Simulation des Roboters mit seiner Umgebung und den Bauteilen auf Kollision getestet. Auf diese Weise wird nun die gesamte Prozessbahn überprüft und bei Erfolg automatisch eine Bahn zum Erreichen dieser berechnet. Nach erfolgreicher Simulation der Roboterbewegungen wird aus diesen entweder gültiger Roboterprogrammiercode generiert oder der Roboter erhält die passenden Befehle zur Laufzeit. Dazu können die notwendigen Befehlsoptionen, wie Geschwindigkeit, die Zonenoption oder das Schalten von Werkzeugen, eingestellt werden.

## Abstract

In this thesis, a robot library with collision detection and configuration determination is developed. The aim is to maneuver an industrial robot, with previously defined process paths, in space. For this purpose its kinematics is analyzed in order to determine possible configurations for reaching a pose. These are in subsequent simulation of the robot, with its environment and the components, testet for collision. In this way, the entire process path is now checked and, if successful, a path is automatically calculated to reach it. After successful simulation of the robots movements, either valid robot programming code is generated, or the robot receives the corresponding commands at runtime. For this, the necessary command options, such as speed, the zone option or tool switching, can be set.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	I
<b>Tabellenverzeichnis</b>	II
<b>Quellcodeverzeichnis</b>	II
<b>Abkürzungsverzeichnis</b>	III
<b>1 Einleitung</b>	<b>1</b>
<b>2 Grundlagen</b>	<b>5</b>
2.1 Vektornormen . . . . .	5
2.2 Posen . . . . .	6
2.3 Affine Transformation . . . . .	9
2.4 Roboterkinematik . . . . .	11
2.4.1 Denavit-Hartenberg . . . . .	11
2.4.2 Mehrdeutigkeiten . . . . .	12
2.4.3 Singularitäten . . . . .	12
2.5 Netzwerkprogrammierung . . . . .	13
2.5.1 UDP-Protokoll . . . . .	14
2.5.2 TCP-Protokoll . . . . .	14
2.5.3 TCP-Nachrichten . . . . .	14
<b>3 Stand der Technik</b>	<b>16</b>
3.1 Software der Roboterhersteller . . . . .	16
3.2 RoboDK . . . . .	17
3.3 V-REP . . . . .	18
3.4 Open Source . . . . .	18
3.4.1 Robotics Library . . . . .	19
3.4.2 OpenRAVE . . . . .	19
3.4.3 Movelt! . . . . .	20
<b>4 Inverse Kinematik</b>	<b>22</b>
4.1 Methodenwahl . . . . .	24
4.2 Analytisches Verfahren mit IKFast . . . . .	25
4.3 Implementierung in Movelt! . . . . .	27
4.4 Abbildung der Roboterkinematik . . . . .	27
4.4.1 Ermittlung der kinematischen Kette . . . . .	28
4.4.2 Ausführung von IKFast . . . . .	30
4.4.3 Auswertung des erzeugten Quellcodes . . . . .	32
<b>5 Kollisionserkennung</b>	<b>35</b>
5.1 Flexible Collision Library . . . . .	35
5.2 CAD-Dateien zur Kollisionserkennung . . . . .	36
5.3 Konvexe Zerlegung . . . . .	36

<b>6 Bahnplanung</b>	<b>38</b>
6.1 Bahnbeschreibung . . . . .	38
6.1.1 Lineare Wegpunkte . . . . .	43
6.1.2 Zirkulare Wegpunkte . . . . .	43
6.1.3 Weginformationen . . . . .	45
6.2 Roboteraufgaben . . . . .	46
6.3 Bewegungsplanung zwischen Bahnen . . . . .	46
6.4 Trajektorie . . . . .	49
6.5 Visualisierung . . . . .	50
<b>7 Roboterschnittstelle</b>	<b>51</b>
7.1 Roboterprogrammiersprache ABB RAPID . . . . .	51
7.2 Generierung von RAPID-Programmen . . . . .	54
7.3 Kommunikation zur Laufzeit . . . . .	57
7.3.1 Nachrichtenprotokoll . . . . .	58
7.3.2 Ausführbare Befehle . . . . .	59
7.3.3 Echtzeitproblematik . . . . .	60
<b>8 Eingabeschnittstelle</b>	<b>62</b>
8.1 Gestaltung einer prozedualen Eingabeschnittstelle . . . . .	62
8.1.1 Optionale Eingabeparameter . . . . .	63
8.1.2 Weginformationen . . . . .	65
8.1.3 Roboteraufgaben . . . . .	66
8.2 Anbindung zu einer interpretierten Programmiersprache . . . . .	66
8.2.1 Die Programmiersprache Python . . . . .	67
8.2.2 Python-Anbindung . . . . .	67
<b>9 Ergebnisse</b>	<b>69</b>
9.1 Einfache Schweißbauteile . . . . .	69
9.2 Barrieren . . . . .	74
9.3 Singularität . . . . .	75
<b>10 Diskussion</b>	<b>77</b>
10.1 Zusammenfassung . . . . .	77
10.2 Auswertung . . . . .	78
10.3 Ausblick . . . . .	79
<b>Literatur</b>	<b>82</b>
<b>A Konfigurationsdateien für OpenRAVE</b>	<b>i</b>
<b>B C++-Programm zum Testen von IKFast</b>	<b>v</b>
<b>C Testprogramm der Roboterschnittstelle</b>	<b>vii</b>
<b>D Testprogramme der Eingabeschnittstelle</b>	<b>xv</b>
<b>E Dokumentation der entwickelten Roboterbibliothek</b>	<b>xviii</b>

## Abbildungsverzeichnis

1	ABB IRB 360 Delta-Roboter [14] . . . . .	1
2	ABB IRB 2600ID Schweißroboter [14] . . . . .	2
3	Schema der Softwaremodule . . . . .	3
4	Schema der Rotationsachse . . . . .	7
5	Schema der affinen Transformation . . . . .	9
6	Denavit-Hartenberg-Notation [13] . . . . .	11
7	Schema einer Mehrdeutigkeit . . . . .	12
8	Schema einer Singularität . . . . .	12
9	Schema des Paketaufbaus einer Netzwerkübertragung [7, S. 15] . . . . .	13
10	Schema der Protokolle TCP und UDP [7, S. 15] . . . . .	14
11	ABB RobotStudio . . . . .	16
12	RoboDK mit Python-Schnittstelle . . . . .	17
13	Coppelia Robotics V-REP [19] . . . . .	18
14	Bahnplanung der Robotics Library [37] . . . . .	19
15	Architektur von MoveIt! [26] . . . . .	20
16	Bahnplanung von MoveIt! . . . . .	21
17	Vereinfachte Darstellung eines Konfigurationsfehlers bei einem Zwei-Achsen-Roboter . . . . .	24
18	Messung der Achse 1 innerhalb RobotStudio . . . . .	28
19	ABB IRB 2600ID in der Visualisierung von OpenRAVE, mit den Achstellungen ( $0^\circ, -20^\circ, 60^\circ, 0^\circ, 60^\circ, 0^\circ$ ) . . . . .	31
20	Alle Lösungen der inversen Kinematik in der OpenRAVE-Visualisierung	32
21	ABB IRB 2600ID in RobotStudio, mit einer Selbstkollision . . . . .	35
22	Vergleich verschiedener Kollisionserkennungen [24] . . . . .	36
23	ABB IRB 2600ID in RobotStudio, mit konvexer Zerlegung der Achsen zur Kollisionserkennung . . . . .	37
24	Schema einer Bahn, bestehend aus drei Wegpunkten . . . . .	42
25	<i>RRT-Connect</i> -Algorithmus bei einer 2D-Problemstellung [8, S. 4] . . . . .	47
26	<i>RRT-Connect</i> -Pseudocode [8, S. 2f] . . . . .	48
27	Visualisierung einer Trajektoriebewegung . . . . .	50
28	Zone zwischen zwei linearen RAPID-Bewegungen [15, S. 24] . . . . .	54
29	Zu frühe Ausführung externer Werkzeuge [15, S. 29] . . . . .	54
30	Endeffektorbewegung bei einem generiertem RAPID-Programmcode mit Wegpunkten aus MoveL und MoveC . . . . .	56
31	Endeffektorbewegung bei einem generiertem RAPID-Programmcode mit Wegpunkten aus MoveAbsJ und einer Schrittgröße von $1\text{ mm}$ . . . . .	57
32	Scheinbar flüssige Bewegung des Roboterendeffektors durch MoveAbsJ mit einer Schrittgröße von $1\text{ mm}$ . . . . .	57
33	Endeffektorbewegung bei der Kommunikation zur Laufzeit . . . . .	61
34	Visualisierung nach Aufruf der show-Prozedur . . . . .	68
35	Endeffektorbewegung einer langen V-Naht, bestehend aus mehreren Schweißnähten . . . . .	70
36	Endeffektorbewegung einer Kehlnaht in horizontaler Überkopf-Position	71
37	Endeffektorbewegung einer inneren Kehlnaht mit einer Schweißnaht .	71
38	Endeffektorbewegung einer inneren Kehlnaht aus mehreren Schweißnähten . . . . .	72

39	Endeffektorbewegung mehrerer Kehlnähte . . . . .	73
40	Endeffektorbewegung zweier Kehlnähte mit Barriere . . . . .	74
41	Endeffektorbewegung einer Kehlnaht mit komplizierter Barriere . . . . .	75
42	Endeffektorbewegung durch eine Singularität . . . . .	76

## Tabellenverzeichnis

1	Achsbegrenzungen des ABB IRB 2600ID . . . . .	33
2	IKFast Ergebnisse zufälliger Konfigurationen des ABB IRB 2600ID . . . . .	34
3	Durchschnittliche Rechenzeiten der Kollisionserkennung . . . . .	37
4	Maximale Achsgeschwindigkeiten des ABB IRB 2600ID . . . . .	40
5	Beispiel einer Standardkonfiguration und der Achsaufwände . . . . .	41
6	Dynamisches Protokoll zur Ausführung von Roboterbefehlen . . . . .	58
7	Nachricht des MoveAbsJ0-Befehls . . . . .	59
8	Datenstruktur zur Übertragung einer Konfiguration . . . . .	59
9	Datenstruktur zur Übertragung einer Pose . . . . .	60
10	Datenstruktur des AddToList0 Befehls . . . . .	61

## Quellcodeverzeichnis

1	Gekürztes Python-Programm des Schweißbauteils . . . . .	67
2	Live-Interpreter-Eingaben zur Darstellung einer Pose . . . . .	68
3	Codeausschnitt der langen V-Naht . . . . .	70
4	Codeausschnitt der Kehlnaht in horizontaler Überkopf-Position . . . . .	71
5	Codeausschnitt der inneren Kehlnaht mit einer Schweißnaht . . . . .	72
6	Codeausschnitt der inneren Kehlnaht aus mehreren Schweißnähten . . . . .	73
7	Codeausschnitt der mehreren Kehlnähte . . . . .	73
8	Codeausschnitt der Barriere mit zwei Kehlnähten . . . . .	74
9	Codeausschnitt der Kehlnaht mit komplizierter Barriere . . . . .	75
10	OpenRave-Umgebungs-XML-Datei eines einzelnen ABB IRB 2600ID . . . . .	i
11	OpenRave-Umgebungs-XML-Datei des ABB IRB 2600ID in einer Schweißumgebung . . . . .	i
12	OpenRave-Roboter-XML-Datei des ABB IRB 2600ID mit Schweißwerkzeug . . . . .	ii
13	OpenRave-Kinematik-XML-Datei des ABB IRB 2600ID . . . . .	iii
14	IKFast-Testprogramm . . . . .	v
15	C++-Programmbeispiel des Schweißbauteils . . . . .	vii
16	Generierter RAPID-Programmcode . . . . .	xi
17	RAPID-Programmcode zur Ansteuerung des Roboters zur Laufzeit . . . . .	xii
18	C++-Programm des Schweißbauteils mit der Eingabeschnittstelle . . . . .	xv
19	Python-Programm des Schweißbauteils . . . . .	xvii

## Abkürzungsverzeichnis

2D .....	Zweidimensional
3D .....	Dreidimensional
CAD .....	<i>Computer Aided Design</i>
CD .....	<i>Compact Disc</i>
DOF .....	<i>Degrees of Freedom</i>
HTML .....	<i>Hypertext Markup Language</i>
mod .....	RAPID Module-Dateiformat
NC .....	<i>Numerical Control</i>
PDF .....	<i>Portable Document Format</i>
PtP .....	<i>Point to Point</i>
RRT .....	<i>Rapidly-exploring Random Tree</i>
SLERP .....	<i>Spherical Linear Interpolation</i>
TCP .....	<i>Transmission Control Protocol</i> oder <i>Tool Center Point</i>
UDP .....	<i>User Datagram Protocol</i>
XML .....	<i>Extensible Markup Language</i>

# 1 Einleitung

Die Verwendung von Industrierobotern steigt stetig an. Laut einer Prognose der International Federation of Robotics sollen im Jahr 2020 mehr als drei Millionen Industrieroboter in Fabriken verwendet werden [25].

Ein Großteil der Industrieroboter wird dazu verwendet, immer gleiche, sich wiederholende Tätigkeiten durchzuführen. Dabei werden die Roboterbewegungen fest eingespielt und auf Signal ausgeführt. Diese Programmierung kann entweder online am Roboter stattfinden, oder offline durch die Abbildung des Prozesses mithilfe von CAD-Dateien in 3D [11].

Sobald die Bewegungen des Roboters nicht planbar sind, steigt der Aufwand der Programmierung erheblich an. Zuerst wird die Bewegung des Roboters benötigt. Diese wird z. B. aus der 2D- oder 3D-Bildverarbeitung mithilfe von Posen generiert (mehr zu Posen, siehe Abschnitt 2.2). Bei der folgenden Bahnplanung müssen auf Mehrdeutigkeiten, Singularitäten und die Kollisionserkennung eingegangen werden (mehr zu Mehrdeutigkeiten und Singularitäten, siehe Abschnitt 2.4.2 und 2.4.3).

Wenn es sich bei dem Prozess um einfache Handhabungen wie dem *Pick and Place* handelt, kann die Konfiguration (Achsstellungen) des Roboters vorbestimmt werden, um somit das Problem der Mehrdeutigkeiten zu beseitigen. Weiterhin werden dort die Roboter so integriert, dass sie über simple Achsbegrenzungen keine Kollision verursachen können. Ein oft verwendeter Industrieroboter hierfür ist ein Parallelroboter wie der Delta-Roboter in Abbildung 1.



Abbildung 1: ABB IRB 360 Delta-Roboter [14]

Problematisch wird es, wenn die Bahnen des Roboters noch komplexer sind. Bei unbekannten Objekten können Kollisionen auftreten, sodass die Bahnen eventuell bestimmte Konfigurationen benötigen, um diese zu vermeiden. Das Schweißen unbekannter Bauteile beinhaltet solch eine Problematik.

Damit diese Prozesse industrietauglich werden, müssen sie einen hohen Abdeckungsgrad erfüllen. Im Beispiel, Schweißen unbekannter Bauteile, sollte der Prozess mit einer hohen Anzahl an verschiedenen Bauteilen, die innerhalb der Rahmenbedingungen liegen, zurecht kommen. Abbildung 2 zeigt einen hierfür verwendbaren Schweißroboter.



**Abbildung 2:** ABB IRB 2600ID Schweißroboter [14]

Innerhalb eines Forschungsprojektes zur automatischen Dekontamination von Bauteilen, wird ein Industrieroboter benötigt, der ebenfalls mit unbekannten Objekten und Bahnen zurecht kommen muss. Diese Bauteile werden zunächst in 3D gescannt. Anschließend werden mit 3D-Bildverarbeitungsprozeduren Bahnen erstellt, die der Roboter, falls möglich, ohne Kollision anfahren sollte.

**Im Rahmen dieser Masterarbeit soll eine Software entwickelt werden, die zuvor generierte Bahnen nutzt, um damit einen Industrieroboter über ein Bauteil kollisionsfrei manövrieren zu können.**

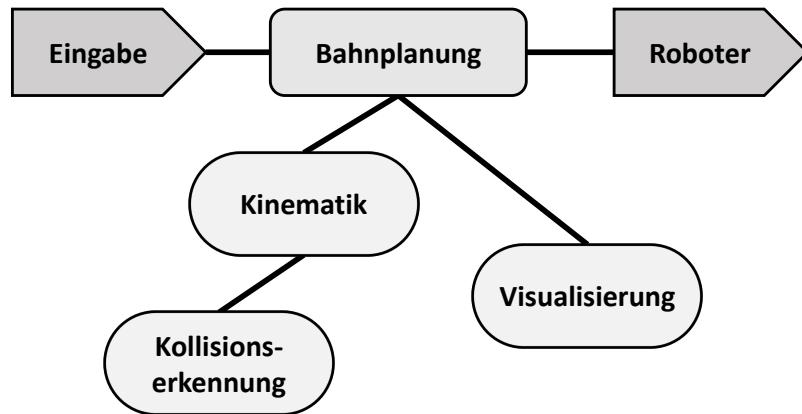
Für das Forschungsprojekt wird zunächst der ABB Roboter IRB 2600ID verwendet, siehe Abbildung 2. Somit zielt diese Arbeit darauf ab, diesen Roboter ansteuern zu können.

Die zu erstellende Software soll universell einsetzbar sein, um zukünftige Projekte schneller realisieren zu können. Oft werden für komplexe Industrieroboter-Aufgabenstellungen, neue Modelle, Datensätze und komplett Programme für diese einzelnen Aufgaben entwickelt. Dabei wird viel Aufwand in etwas gesteckt, was oft wiederholt wird [4, S. 3].

Für die Realisierung dieser Aufgabe gibt es zwar schon Komplettlösungen, diese sind aber nicht offen und in einigen Bereichen schwer auf die Problemstellung anpassbar. Da die Hersteller den Quellcode nicht vergeben, ist eine Anpassung an deren Schnittstellen und Vorgaben gebunden und somit eine Kompromisslösung.

Deswegen wird eine eigene offene Software entwickelt, die für dieses und kommende Projekte nutzbar sein wird.

In Abbildung 3 sind die Bestandteile der Software schematisch abgebildet:



**Abbildung 3:** Schema der Softwaremodule

- **Eingabeschnittstelle**

Um den Industrieroboter zu programmieren, wird eine eigene Roboterbibliothek benötigt, die roboterunabhängig und genau auf die Software zugeschnitten ist. Um Programmierfreiheiten zu bewahren, wird sie als Programmierbibliothek in der Sprache C++ umgesetzt.

- **Kinematik**

Beinhaltet Funktionen zum Berechnen der Vorwärtsskinematik und der inversen Kinematik. Dazu enthält dieses Modul alle kinematischen Informationen des Roboters, wie Achsabstände, maximale Achswinkel und maximale Achsgeschwindigkeiten.

- **Kollisionserkennung**

Damit der Roboter nicht mit sich selbst oder der Umgebung kollidiert, wie beispielsweise mit einem Schaltschrank, einem Arbeitstisch oder einem Bauteil, wird die Kollisionserkennung benötigt. Nach Berechnen einer Achskonfiguration aus der inversen Kinematik wird sie anschließend mit diesem Modul überprüft.

- **Bahnplanung**

Aus der Eingabeschnittstelle nimmt sich die Bahnplanung die abstrakten Informationen, wie die Bahn aussieht. Zusammen mit der inversen Kinematik und der Kollisionserkennung, wird die endgültige Bahn bestimmt. Eine Bahn wird dabei aus Geraden, Kurven, etc. beschrieben und in der inversen Kinematik mit einstellbarer Schrittweite auf Erreichbarkeit und Kollision getestet.

- **Visualisierung**

Um die Ergebnisse der Bahnplanung visuell testen zu können, wird der Roboter und die Umgebung durch deren CAD-Dateien in einem Programmfenster dargestellt.

- **Roboterschnittstelle**

Die erzeugte Bahn aus der Bahnplanung wird abschließend entweder in einem Postprozessor in ein roboterabhängiges Programm umgewandelt (z. B. ABB RAPID, siehe Abschnitt 7.1), oder dem Roboter wird zur Laufzeit gesendet, welche Bewegungen und Befehle er ausführen soll.

Um eine höhere Flexibilität zu erreichen, wird die zu erstellende Roboterbibliothek zwischen dem eigentlichem Prozess und dem Weg dorthin unterscheiden. Am Beispiel eines Schweißbauteils mit mehreren Schweißnähten, können diese so jeweils mit einer eigenen Bahn beschrieben werden. Zwischen den Bahnen benötigen die Wege oft keine genaue Definition. Dann genügt es, wenn sie ohne Kollision erreicht werden können. Wenn im Prozess eine weitere externe Achse, beispielsweise ein Drehtisch, vorhanden ist, kann mit dieser Technik ein höherer Abdeckungsgrad erreicht werden. Für den Fall, dass eine der Schweißnahtbahnen nicht ohne Kollision erreichbar ist, wird sie mit einer anderen Einstellung der externen Achse geprüft.

## 2 Grundlagen

In diesem Kapitel werden einige, für diese Arbeit notwendige, Grundlagen erläutert.

### 2.1 Vektornormen

Genauso wie der Betrag eines Skalars  $|x|$ , beschreibt eine Vektornorm die Größe des Vektors. Dabei ist unerheblich, wie viele Dimensionen dieser Vektor besitzt. Eine bekannte Vektornorm ist der geometrische Abstand in der 2. und 3. Dimension, siehe Gleichung 2.1.

$$d_{xy} = \sqrt{\Delta_x^2 + \Delta_y^2} \quad d_{xyz} = \sqrt{\Delta_x^2 + \Delta_y^2 + \Delta_z^2} \quad (2.1)$$

Für n-dimensionale Vektoren gibt es verschiedene Arten von Normen. Besonders Software kann davon profitieren. Wenn die genaue geometrische Länge des Vektors nicht benötigt wird, kann mit anderen Normen Rechenzeit gespart werden. Beispielsweise benötigt die Berechnung der Wurzel des geometrischen Abstandes einige Rechenzeit.

Eine allgemeine Form vieler Vektornormen ist die *p-Norm*, siehe Gleichung 2.2.

$$\|x\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{\frac{1}{p}} \quad (2.2)$$

Das Einsetzen von  $p = 2$  ergibt die *euklidische Norm*, die den geometrischen Abstand darstellt, siehe Gleichung 2.3.

$$\|x\|_2 = \sqrt{\sum_{i=1}^n |x_i|^2} \quad (2.3)$$

Die *Summennorm*, siehe Gleichung 2.4, summiert lediglich die Beträge der Vektorkomponenten. Sie ergibt sich durch Einsetzen von  $p = 1$ .

$$\|x\|_1 = \sum_{i=1}^n |x_i| \quad (2.4)$$

Eine weitere besondere Vektornorm ist die  $\infty$ -Norm, auch Maximumsnorm genannt. Sie beschreibt den größten Betrag der Vektorkomponenten, siehe Gleichung 2.5.

$$\|x\|_\infty = \max_{i=1,\dots,n} |x_i| \quad (2.5)$$

Angenommen der Vektor  $x$  hat die Komponenten  $(10, -3, 7)$ , zeigen folgende Beispiele die Ergebnisse der vorgestellten Vektornormen:

$$x = (10, -3, 7)$$

$$\|x\|_1 = 20 \quad \|x\|_2 = 12,57 \quad \|x\|_\infty = 10$$

## 2.2 Posen

Im euklidischen Raum  $\mathbb{R}^3$  beschreibt die Kombination aus Position und Orientierung eine Pose [20, S. 12]. Sie wird u. a. dafür genutzt, den Endeffektor (Werkzeug [20, S. 9]) eines Industrieroboters zu bewegen.

Die Position, bzw. Translation, wird mithilfe eines Vektors dargestellt, siehe Gleichung 2.6.

$$trans = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad (2.6)$$

Für die Orientierung gibt es verschiedene Darstellungen:

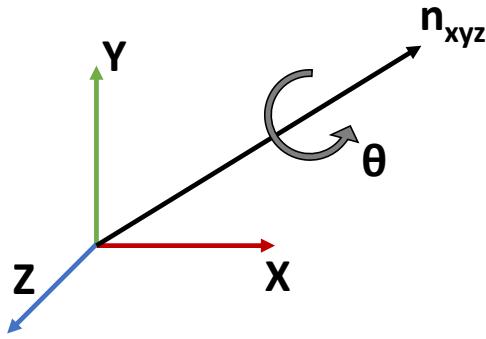
- **Eulersche Winkel**

Bei dieser Rotationsart wird nacheinander um die  $X$ ,  $Y$  und  $Z$ -Achse rotiert. Ihr Vorteil liegt in der Lesbarkeit. Nachteile dieser Form sind zum einen Singularitäten, da Orientierungen existieren, bei denen es mehrere oder unendlich viele Lösungen für die Rotationswerte gibt. Des Weiteren gibt es verschiedene Implementierungen für die Rotationsreihenfolge (u. a.  $X \rightarrow Y \rightarrow Z$  und  $Z \rightarrow Y \rightarrow X$ ). Es werden drei Elemente benötigt, siehe Gleichung 2.7.

$$rot_{euler} = \begin{pmatrix} x_{rot} \\ y_{rot} \\ z_{rot} \end{pmatrix} \quad (2.7)$$

- **Rotationsachsen**

Eine ebenfalls einfache Darstellung der Orientierung ist ein Vektor der Rotationsachse  $n_{xyz}$ , zusammen mit der Drehung  $\theta$ , siehe Abbildung 4. Im Gegensatz zu den eulerschen Winkeln ist diese Form der Rotation eindeutig.



**Abbildung 4:** Schema der Rotationsachse

Für den Vektor der Rotationsachse  $n_{xyz}$  wird üblicherweise ein normierter Vektor genutzt ( $\|n_{xyz}\|_2 = 1$ ). Es werden vier Elemente benötigt, siehe Gleichung 2.8.

$$rot_{rotaxis} = \begin{pmatrix} n_x \\ n_y \\ n_z \\ \theta \end{pmatrix} \quad (2.8)$$

Alternativ können durch Multiplikation der einzelnen Normvektorkomponenten  $n_x$ ,  $n_y$  und  $n_z$ , mit der Drehung  $\theta$ , die benötigten Elemente auf drei begrenzt werden, siehe Gleichung 2.9. Die Drehung wird nun durch die euklidische Norm bestimmt.

$$rot_{rotaxis} = \begin{pmatrix} n_x \cdot \theta \\ n_y \cdot \theta \\ n_z \cdot \theta \end{pmatrix} \quad (2.9)$$

- **Quaternionen**

Wie die komplexen Zahlen, erweitern sie die reellen Zahlen auf eine höhere Dimension, im diesem Fall auf vier, siehe Gleichung 2.10.

$$q = \begin{pmatrix} a \\ b \cdot i \\ c \cdot j \\ d \cdot k \end{pmatrix} \quad (2.10)$$

Für Orientierungen werden die Quaternionen ähnlich zu den Rotationsachsen aufgebaut, siehe Gleichungen 2.11 - 2.13. Im Gegensatz zu diesen, hat die Quaternionendarstellung den Vorteil, Grundrechenregeln zu besitzen. Ihr Nachteil liegt in der schlechten Lesbarkeit. Es werden ebenfalls vier Elemente benötigt, siehe Gleichung 2.13.

$$C = \cos\left(\frac{\theta}{2}\right) \quad (2.11)$$

$$S = \sin\left(\frac{\theta}{2}\right) \quad (2.12)$$

$$rot_{quat} = \begin{pmatrix} C \\ n_x \cdot S \cdot i \\ n_y \cdot S \cdot j \\ n_z \cdot S \cdot k \end{pmatrix} \quad (2.13)$$

- **Rotationsmatrizen**

Orientierungen können auch mithilfe von Matrizen dargestellt werden. So eine Matrix besteht dabei aus drei normierten Spaltenvektoren  $X_{xyz}$ ,  $Y_{xyz}$  und  $Z_{xyz}$ , wobei diese Vektoren auf die Richtung der  $X$ ,  $Y$  und  $Z$  Achse der Pose zeigen. Mehr dazu im folgendem Abschnitt 2.3.

Eine Drehung um die Z-Achse sieht dementsprechend wie folgt aus:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Eine Rotationsmatrix benötigt neun Elemente, siehe Gleichung 2.14.

$$rot_{mat} = \begin{bmatrix} X_x & Y_x & Z_x \\ X_y & Y_y & Z_y \\ X_z & Y_z & Z_z \end{bmatrix} \quad (2.14)$$

## 2.3 Affine Transformation

Um das Koordinatensystem von Positionen oder Posen im Raum zu wechseln, werden affine Transformationen angewendet.

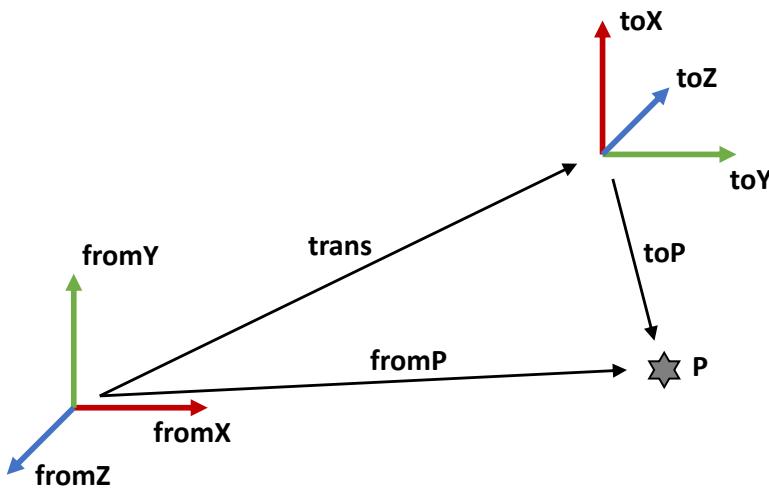


Abbildung 5: Schema der affinen Transformation

In Abbildung 5 sind zwei Koordinatensysteme dargestellt: *from* und *to*. Um das Koordinatensystem von Punkt  $P$  von *from* nach *to* zu wechseln, wird die affine Transformationsmatrix benötigt.

Die zugrunde liegende Matrix besteht aus einer  $3 \times 3$  Rotationsmatrix und einem  $1 \times 3$  Spaltenvektor für die Translation. Damit die Matrix quadratisch und invertierbar ist, wird eine vierte Zeile mit dem Inhalt  $[0 \ 0 \ 0 \ 1]$  hinzugefügt.

$$T = \begin{bmatrix} rot & trans \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} X_x & Y_x & Z_x & trans_x \\ X_y & Y_y & Z_y & trans_y \\ X_z & Y_z & Z_z & trans_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.15)$$

Gleichung 2.15 zeigt den Aufbau der Transformationsmatrix. Der Spaltenvektor *trans* stellt die Translation zwischen den beiden Koordinatensystemen dar. Die Rotationsmatrix *rot* besteht aus den drei Spaltenvektoren *X*, *Y* und *Z*. Diese Vektoren beinhalten jeweils die normierte Richtung des *X*-, *Y*- und *Z*-Vektors des anderen Koordinatensystems. Eine einfache Translation, ohne Rotation, ergibt sich dementsprechend wie folgt:

$$TransMat = \begin{bmatrix} 1 & 0 & 0 & trans_x \\ 0 & 1 & 0 & trans_y \\ 0 & 0 & 1 & trans_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Transformationsmatrix aus Abbildung 5, welche Punkt  $P$  vom Koordinatensystem  $from$  nach  $to$  transformiert, sieht wie folgt aus:

$$M = \begin{bmatrix} 0 & 1 & 0 & trans_x \\ 1 & 0 & 0 & trans_y \\ 0 & 0 & -1 & trans_z \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

Hier zeigt aus der Perspektive von  $from$  der normierte  $X$ -Vektor von  $to$  nach  $Y$ , der  $Y$ -Vektor zeigt nach  $X$  und der  $Z$ -Vektor nach  $-Z$ . Zu beachten ist die abschließende Invertierung der Matrix, da sonst der Punkt  $P$  aus dem Koordinatensystem  $to$  nach  $from$  transformiert wird. Durch die Invertierung lässt sich die Transformationsrichtung umkehren.

Damit ein Punkt, bzw. eine Position, mit einer affinen Transformationsmatrix multiplizierbar ist, wird ein Spaltenvektor mit vier Elementen benötigt. Die ersten drei stellen den Punkt dar, gefolgt von einer 1, siehe Gleichung 2.16.

$$P = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad (2.16)$$

Zur Transformation des Punktes  $P$  vom Koordinatensystem  $from$  nach  $to$  wird dieser lediglich mit der passenden affinen Transformationsmatrix multipliziert, siehe Gleichung 2.17. Dabei ist die Reihenfolge  $Matrix \cdot Punkt$  einzuhalten.

$$P_{to} = M \cdot P_{from} \quad (2.17)$$

Eine Pose kann genauso wie ein Koordinatensystem dargestellt werden. Um eine Pose von einem zum anderem Koordinatensystem zu transformieren, wird dieses durch eine Matrixmultiplikation mit der Transformationsmatrix multipliziert.

Angenommen es gibt die Koordinatensysteme  $A$ ,  $B$  und  $C$ , ist es möglich, dessen Transformationsmatrizen zu multiplizieren, um so einen Punkt direkt von  $A$  nach  $C$  zu transformieren, statt von  $A$  über  $B$  nach  $C$ .

## 2.4 Roboterkinematik

Die Beziehung zwischen der Roboterendeffektorbewegung (Endeffektor = Werkzeug des Roboters [20, S. 9]) im Raum und dessen Konfiguration (Zusammenfassung aller Achsstellungen [20, S. 8]) beschreibt die Roboterkinematik. Zum einen lässt sich mit ihr aus der Vorwärtskinematik und den Achsstellungen, die Pose des Endeffektors bestimmen. Dazu genügt ein geometrischer Ansatz, indem Achse für Achse die einzelnen Gelenkposen, bis hin zur Endeffektorpose bestimmt werden. Des Weiteren wird sie für die inverse Kinematik benötigt. Diese bestimmt die möglichen Achsstellungen für eine vorgegebene Pose des Endeffektors [20, S. 12].

### 2.4.1 Denavit-Hartenberg

Zur Abbildung der kinematischen Kette, werden affine Transformationsmatrizen zwischen den Roboterachsen benötigt. So kann beispielsweise für die Vorwärtskinematik von der Roboterbasis aus, jeweils zur nächsten Achse hintransformiert werden, wobei anschließend, abhängig von der Achsstellung, eine Rotation bzw. Translation hinzugefügt wird. Nach der Notation von Denavit und Hartenberg [3], zeigt die  $Z$ -Achse einer Roboterachse stets in Richtung der Rotationsachse, bzw. der Translationsrichtung. Somit wird die Roboterachsstellung durch eine Rotation, bzw. Translation in  $Z$ -Richtung beschrieben.

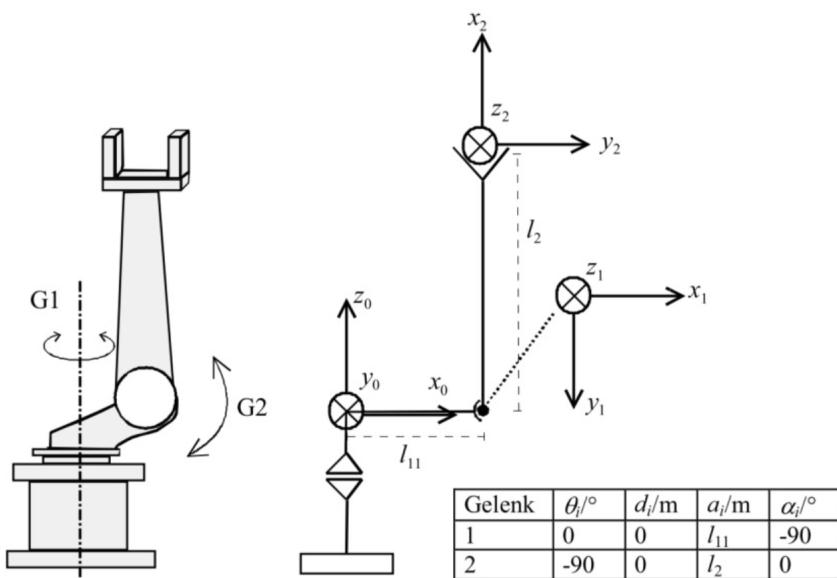


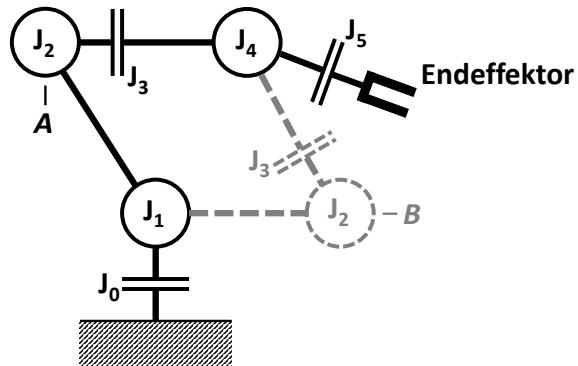
Abbildung 6: Denavit-Hartenberg-Notation [13]

Abbildung 6 zeigt einen Drei-Achsen-Knickarm-Roboter. Die Koordinatensysteme sind, wie beschrieben, so ausgerichtet, dass die  $Z$ -Achse auf der Rotationsachse liegt. Da zur Beschreibung der Transformation zwischen den Achsen auf diese Weise nur vier Parameter nötig sind, beschränkt sich die Denavit-Hartenberg-Notation

auf die Drehung der  $Z$ -Achse  $\theta$  und die Translation in  $Z$ -Richtung  $d$ . Nach dieser Transformation folgt die Translation in  $X$ -Richtung  $a$  und die Rotation um die  $X$ -Achse  $\alpha$ .

#### 2.4.2 Mehrdeutigkeiten

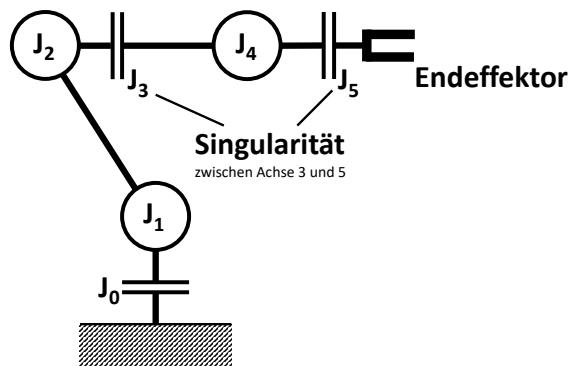
Für die meisten erreichbaren Posen, gibt es mehrere Lösungen der inversen Kinematik. Durch das Vorhandensein von Mehrdeutigkeiten muss sich die Software für eine Lösung entscheiden. In Abbildung 7 ist ein Sechs-Achsen-Knickarmroboter schematisch dargestellt, wobei es in der 2D-Darstellung bereits zwei Lösungen für die Endeffektorpose gibt, nämlich  $A$  und  $B$ .



**Abbildung 7:** Schema einer Mehrdeutigkeit

#### 2.4.3 Singularitäten

Wenn sich zwei Roboterachsen auf der gleichen Rotationsachse befinden, gibt es eine unendliche Anzahl an Lösungen. Abbildung 8 zeigt ein Beispiel, mit einer Singularität zwischen Achse 3 und 5. Wird nun Achse 3 um  $10^\circ$  gedreht, kann Achse 5 auf  $-10^\circ$  gestellt werden, um dieselbe Pose zu erreichen.

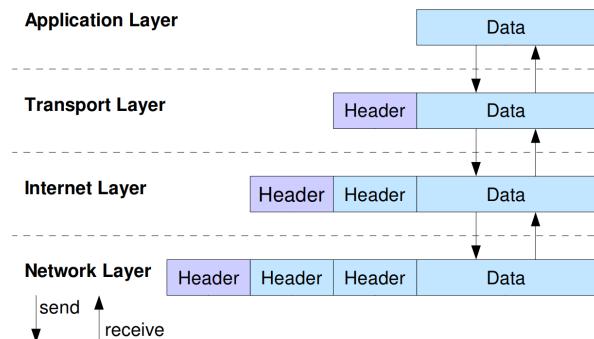


**Abbildung 8:** Schema einer Singularität

Neben dieser gibt es noch weitere Arten von Singularitäten. Wenn der Roboter solch einen Punkt durchfährt, werden durch die inverse Kinematik unendlich schnelle Achsgeschwindigkeiten generiert, um die Endeffektorgeschwindigkeit beizubehalten. Da die Achsgeschwindigkeiten des Roboters durch die Steuerung begrenzt werden, leidet darunter die Endeffektorgeschwindigkeit, was sich z. B. bei einem Lackiervorgang schlecht auf das Ergebnis auswirken kann. Wenn ein ABB Roboter mittels Linearbewegung in die Nähe einer Singularität gerät, wird die Bewegung gestoppt.

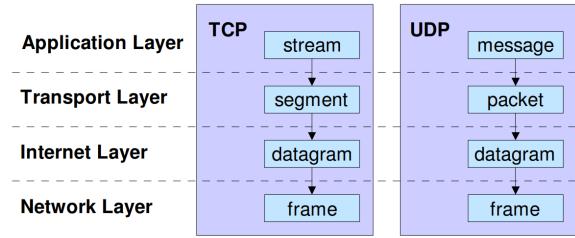
## 2.5 Netzwerkprogrammierung

Eine Art der Übertragung von Daten zwischen zwei Computern ist die Verwendung der Netzwerkschnittstelle. Durch sie ist es möglich, die beiden Computer direkt miteinander zu verbinden, oder die Übertragung über das Netzwerk stattfinden zu lassen. Um eine Abstraktion der Hardware zu ermöglichen, werden standardisierte Protokolle verwendet. Diese hängen an die zu übertragenden, noch weitere Daten an, um so Pakete durch das Netzwerk senden zu können, siehe Abbildung 9. Bei diesen Daten handelt es sich um die Informationen des Zielsystems, gegeben durch u. a. der IP-Adresse und der verwendeten Port Nummer der Verbindung.



**Abbildung 9:** Schema des Paketaufbaus einer Netzwerkübertragung [7, S. 15]

Über das Netzwerk selbst werden die Daten in Paketen aus mehreren Bytes übertragen. Um den Verbindungsaufbau zu vereinfachen, werden die Übertragungsprotokolle UDP und TCP verwendet (*User Datagram Protocol* und *Transmission Control Protocol*). Wie in Abbildung 10 erkennbar ist, definieren die beiden Protokolle die Transportschicht der Datenpakete. Die beiden Computer der Verbindung verarbeiten dann jeweils diese Transportschicht der Übertragung ab und liefern der Anwendung die reinen, empfangenen Daten.



**Abbildung 10:** Schema der Protokolle TCP und UDP [7, S. 15]

### 2.5.1 UDP-Protokoll

Bei dieser Verbindung werden Nachrichten definierter Größe in einer Richtung übertragen. Die Größe der Nachricht ist durch die Internet- und Netzwerkschicht begrenzt, da es die gesamte Nachricht in einem Netzwerkpacet unterbringt. Bei dieser Übertragungsart wird nicht kontrolliert, ob die Nachricht empfangen wird oder ob die Reihenfolge der Nachrichten eingehalten wird. Dadurch ist der Verbindungsauflauf einfach und die Übertragungsgeschwindigkeit höher als die des TCP-Protokolls. Ein weiterer Vorteil liegt in dem einfachen Definieren der übertragenden Nachrichten. Da die Reihenfolge und die Positionen der Bytes innerhalb einer Nachricht eingehalten werden, kann eine Struktur auf die Nachricht angewandt werden. Zur Übertragung einer Position im Raum, könnten beispielsweise die ersten vier Bytes die *X*-Position, die nächsten vier die *Y*-Position und die letzten die *Z*-Position beschreiben. UDP-Nachrichten werden z. B. für Zeitserver verwendet, da hierbei eine Nachricht genügt und der Server somit weniger Aufwand zur Verbindung betreiben muss.

### 2.5.2 TCP-Protokoll

Diese Verbindungsart überträgt Daten in beide Richtungen. Beide Computer können senden und empfangen. Im Gegensatz zu den UDP-Nachrichten, werden alle Netzwerkpakete kontrolliert, somit erhalten die Anwendungen einen *Stream* von Bytes. Diese kommen in der richtigen Reihenfolge an, bis die Verbindung unterbrochen wird. Die TCP-Verbindung eignet sich somit für verlustfreie Übertragungen.

Mehr zum TCP/IP-Protokoll: [7]

### 2.5.3 TCP-Nachrichten

Da bei der TCP-Verbindung mit Byte-*Streams* gearbeitet wird, werden eigene Protokolle zur Definition von Nachrichten benötigt. Einige grundlegende Protokolldefinitionen werden im folgenden kurz beschrieben:

- **Ein Byte als Nachricht**

Das einfachste Protokoll besteht in einer Nachrichtengröße von einem Byte. So kann der *Stream*, Byte für Byte ausgelesen und ausgewertet werden. Der Vorteil liegt in der sehr einfachen Implementierung, der Nachteil in der begrenzten Nachrichtenkomplexität von maximal 256 verschiedenen Möglichkeiten. Beispiele für dieses Protokoll sind einfache Zustandssensoren.

- **Definierte Nachrichtenlänge**

Um mehr Informationen in die Nachricht hinzuzufügen, kann die Nachrichtenlänge vergrößert werden. Der Empfänger liest für eine Nachricht solange Bytes aus dem *Stream*, bis die Nachrichtenlänge erreicht wird. So kann die Nachricht auch komplexere Informationen, wie ganze Roboterposen, beinhalten. Der Vorteil liegt in der einfachen Implementierung. Sobald die eigentliche Nachrichtengröße dynamisch wird, muss bei diesem Protokoll die Nachrichtenlänge die maximale Nachrichtengröße betragen. Der Nachteil liegt in diesem Fall in der Übertragung nicht verwendeter Bytes kürzerer Nachrichten. Ein Beispiel hierfür ist eine aus einem Befehlsbyte und einer folgenden Pose bestehenden Nachricht. Wenn das Befehlsbyte eine Bewegungsprozedur ausführen soll, wird die entsprechende Pose aus der Nachricht entnommen. Wenn hingegen das Befehlsbyte ein Werkzeug aktiviert, wird die Pose ignoriert, muss aber dennoch übertragen werden.

- **Dynamische Nachrichtenlänge mit Terminierungsbyte**

Für eine wechselnde Nachrichtengröße, kann ein Protokoll verwendet werden, dass dem C-String ähnelt. Der Empfänger liest solange Bytes aus dem *Stream*, bis eine Null empfangen wurde. Das hat den Nachteil, dass die Nachricht selbst, keine Null als Bytewert beinhalten darf. Um dennoch Zahlen zu versenden, können diese z. B. in ihrer ASCII-Text-Darstellung übertragen werden (der Wert der ASCII 0 beträgt 48). Durch diese Maßnahmen steigt der Implementierungsaufwand.

- **Vorbestimmte dynamische Nachrichtenlänge**

Ein weiteres dynamisches Protokoll besteht darin, zuerst die Nachrichtenlänge als Zahl zu übertragen. Der Empfänger liest z. B. erst ein Byte, welches die Größe der Nachricht von 0 – 255 darstellt. Daraufhin liest er dementsprechend die genaue Anzahl an Bytes der Nachrichtenlänge aus dem *Stream*. Der Vorteil dieses Protokolls liegt darin, auch mit Nullen zurecht zu kommen. Weiterhin ist die Übertragungsgeschwindigkeit höher, da nicht jedes Byte einzeln überprüft werden muss.

### 3 Stand der Technik

Im Bereich der Roboterprogrammierung, Simulation und Kollisionserkennung gibt es bereits zahlreiche Anwendungen von verschiedenen Herstellern und aus freier Software.

#### 3.1 Software der Roboterhersteller

Die Industrieroboterhersteller bieten im Regelfall eine eigene Software zur Programmierung ihrer Roboter an. Darunter fällt auch der für das Forschungsprojekt genutzte ABB Roboter IRB 2600ID, mit der Software ABB RobotStudio.

Innerhalb von RobotStudio lassen sich die Roboter, neben der Programmierung von Hand in der Programmiersprache RAPID, auch visuell in einer 3D-Simulation programmieren, siehe Abbildung 11. Mithilfe dieser 3D-Rekonstruktion lassen sich ebenfalls Kollisionen mit der Umgebung und dem Roboter selbst umgehen.

Diese Softwareprogramme sind aber im Normalfall dafür ausgelegt, den Roboter einmalig von Hand innerhalb der Software zu programmieren. Selbst wenn die Möglichkeit geboten wird, solch eine Simulationssoftware aus einer externen eigenen Software anzusprechen, ist die Programmierung durch die Software begrenzt und somit nur schwer erweiterbar. Mit den Schnittstellen zu ABB RobotStudio lässt sich beispielsweise nicht die aktuelle Roboterkonfiguration einstellen. Des Weiteren wäre für einen anderen Roboter, anderen Herstellers, eine neue aufwendige Implementierung nötig.

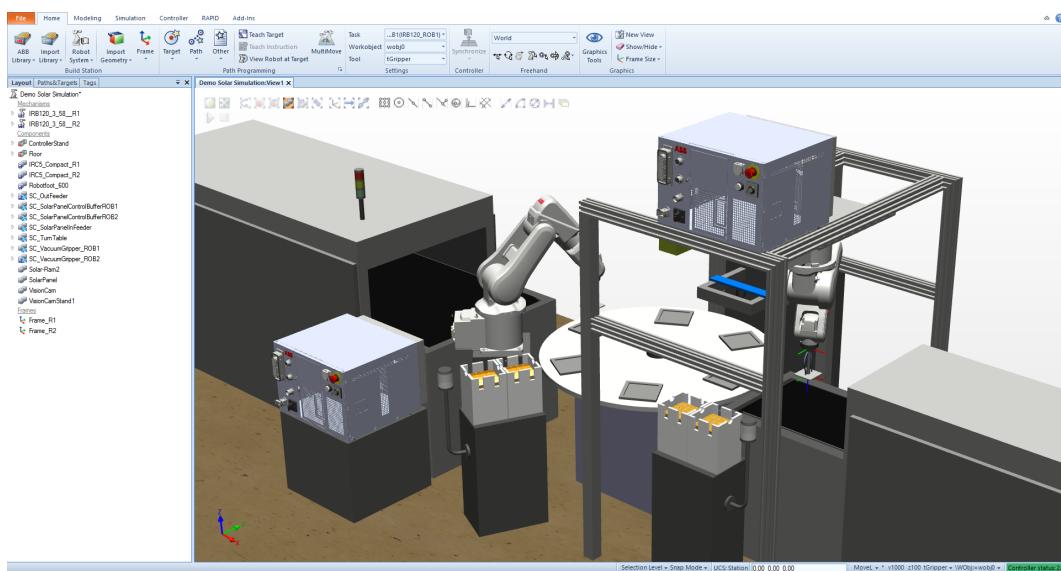


Abbildung 11: ABB RobotStudio

Ohne Schnittstelle zum Simulationsprogramm, lässt sich alternativ über eine Netzwerkverbindung mit der Steuerung und dessen Roboterprogramm kommunizieren.

Über entsprechende Programmierung des Roboters, kann dieser auch zur Laufzeit bestimmen, welche Posen er anfahren kann. Diese Art der Roboterprogrammierung ist ebenfalls durch die Roboterprogrammiersprache begrenzt und zudem aufwendig und herstellerabhängig.

In der Roboterprogrammiersprache ABB RAPID lässt sich zur Laufzeit die inverse Kinematik mithilfe der Funktion CalcJointT berechnen [16, S. 786]. Zur eindeutigen Bestimmung der Konfigurationen, muss bei den Posen allerdings für einige Achsen angegeben werden, in welchen Quadranten sie sich befinden.

## 3.2 RoboDK

Die kommerzielle Industrierobotersoftware RoboDK [36] beinhaltet eine roboterunabhängige Programmierung. Innerhalb einer 3D-Visualisierung lassen sich viele vordefinierte Roboter verschiedener Hersteller einbinden und ausrichten. Neben den Robotern können auch CAD-Dateien von der Umgebung, der Arbeitsplatte und von Werkstücken eingeladen werden. Diese werden zur Berechnung der Kollisionserkennung benötigt. Mithilfe von CAD-Dateien der Werkstücke, können die Roboter auch ohne Programmierung von Hand programmiert werden, z. B. lässt sich so ein Roboter wie ein 3D-Drucker betreiben. RoboDK kann den Roboter entweder zur Laufzeit steuern oder es wird ein roboterabhängiges Programm für die Robotersteuerung mithilfe des Postprozessors erstellt. Für komplexere Aufgaben, wird der Roboter mithilfe von Programmierschnittstellen programmiert, u. a. in der Programmiersprache Python, siehe Abbildung 12.

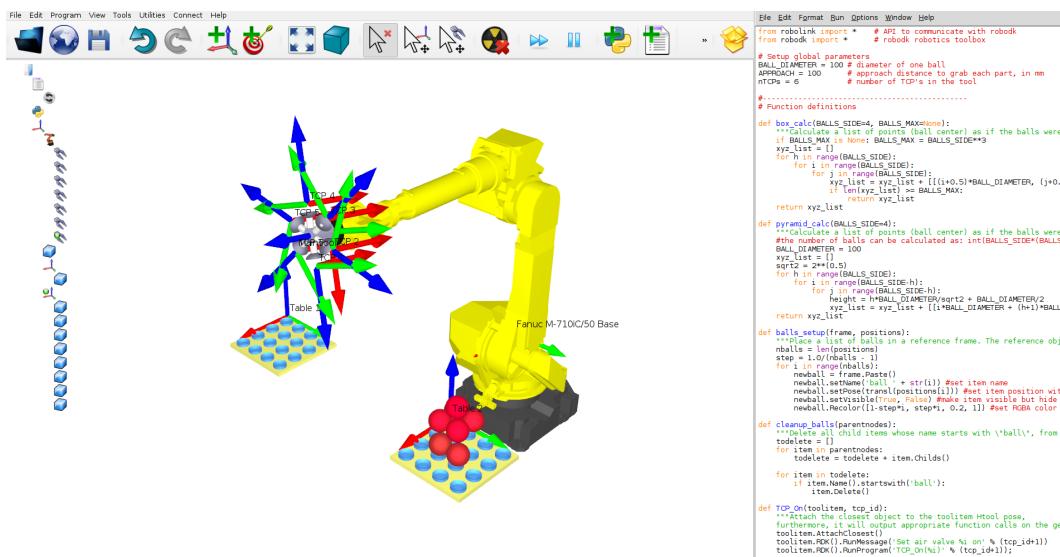


Abbildung 12: RoboDK mit Python-Schnittstelle

Über die Schnittstellen werden Kollisionen getestet, inverse Kinematiken berechnet und der Roboter kann über Wegbefehle angesteuert werden. Sobald der Roboter

über die Wegbefehle angesteuert wird, befindet er sich im Modus ohne Konfigurationsüberwachung. Das heißt, dass der Roboter versucht, seine aktuelle Konfiguration beizubehalten (Für ABB RAPID: [16, S. 59]).

Der Nachteil von RoboDK liegt in der schlechten Erweiterbarkeit, da die Funktionen vorgegeben sind. Weiterhin fehlt dem RoboDK eine Bahnplanung, die die Roboterkonfigurationen berücksichtigt und nicht einfach durch die Startkonfiguration ignoriert.

### 3.3 V-REP

Eine weitere Software zur Simulierung von Industrierobotern ist V-REP von Coppelia Robotics [19]. Einige der Komponenten sind die inverse Kinematik, Kollisionserkennung, Visualisierung, Bahnplanung, Simulation von Sensoren und die Dynamiksimulation. V-REP ist für Windows, MacOS X und Linux verfügbar und kann über eine Softwareschnittstelle in verschiedenen Programmiersprachen genutzt werden, u. a. in C, C++, Java, Python, Matlab und GNU Octave. Neben der kommerziellen Version, bietet Coppelia Robotics auch freie Varianten mit offenem Quellcode an.

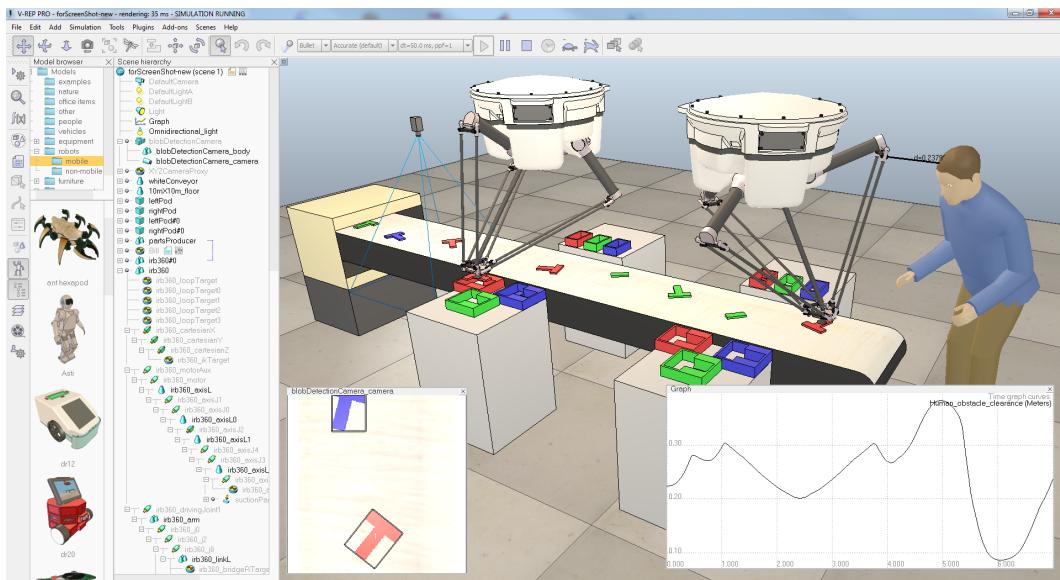


Abbildung 13: Coppelia Robotics V-REP [19]

### 3.4 Open Source

Zur leichteren Implementierung und Automatisierbarkeit empfiehlt es sich, quell-offene Projekte in Betracht zu ziehen. Aus diesen lassen sich, je nach Lizenz, Teile der Software entnehmen. Des Weiteren kann der Programmcode auf Fehler untersucht und erweitert werden.

### 3.4.1 Robotics Library

Die offene C++-Bibliothek Robotics Library [37] ermöglicht durch die Abbildung der Kinematik, einer Bahnplanung und Kollisionserkennung, die Simulation von Industrierobotern. Zur Beschreibung der kinematischen Kette wird eine XML-Datei des Roboteraufbaus benötigt. Durch diese und der Beschreibung der Umgebung, lässt sich mithilfe der C++-Bibliothek die Vorwärtskinematik, die inverse Kinematik und die Kollisionserkennung berechnen. Außerdem hilft ein Modul zur kollisionsfreien Bahnplanung, siehe Abbildung 14.

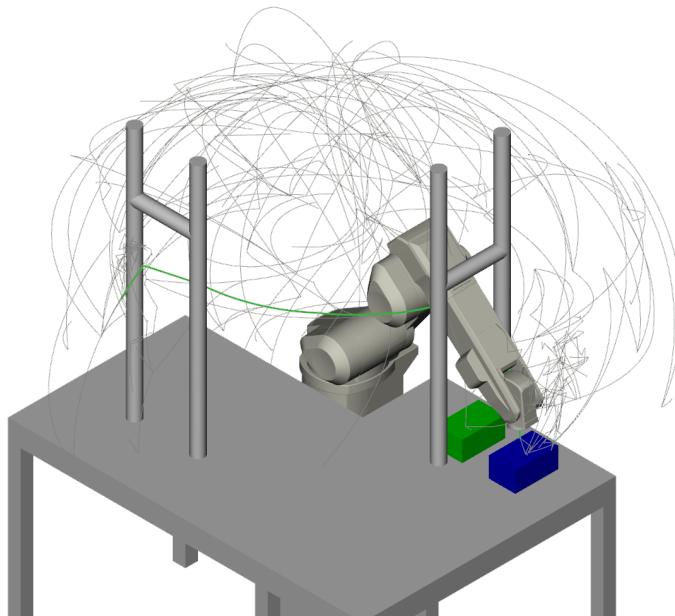


Abbildung 14: Bahnplanung der Robotics Library [37]

### 3.4.2 OpenRAVE

Das offene Projekt OpenRAVE (*Open Robotics Automation Virtual Environment*) [32] hat das Ziel, die Simulation und Analyse der Kinematik und Geometrie verschiedenster Roboter zu vereinfachen. Es beinhaltet Bibliotheken zur Berechnung der Vorwärtskinematik, der inversen Kinematik und der Kollisionserkennung. Die Roboter lassen sich hierzu ebenfalls mithilfe einer XML-Datei anhand ihrer Geometrie und kinematischen Kette beschreiben. Durch die quellöffentliche Bibliothek lassen sich anschließend in den Programmiersprachen C++ oder Python die Berechnungen für die inverse Kinematik und die Kollisionserkennung durchführen. Der Vorteil an OpenRAVE liegt in der einfachen Erweiterbarkeit und Nutzbarkeit.

### 3.4.3 MoveIt!

Diese C++-Bibliothek ist ein Softwarepaket von ROS-Industrial, welches dem Robotik-Framework ROS (*Robot Operating System*) angehört. ROS ist eine offene Plattform zur Programmierung von Robotern und besteht aus vielen kleinen und großen Programmpaketen, u. a. für die Visualisierung und Bewegungsplanung im Raum. ROS-Industrial ergänzt diese Pakete zur Nutzung von Industrierobotern für die Herstellung und Automatisierung in der Industrie [38].

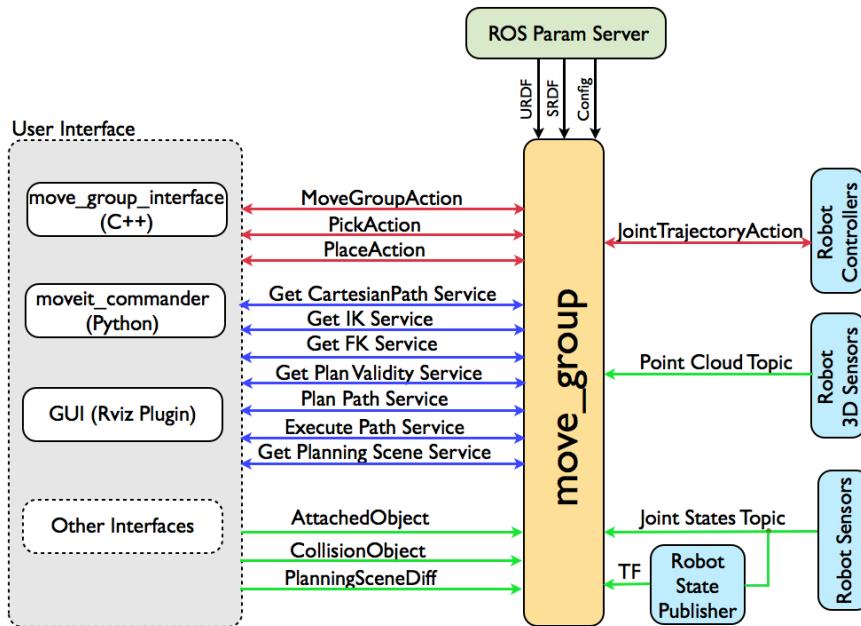
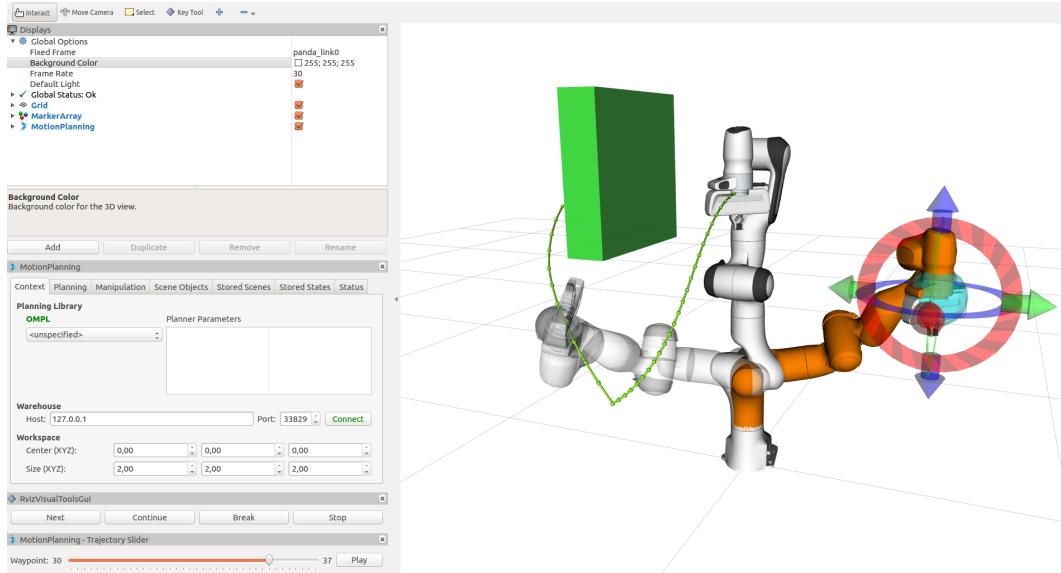


Abbildung 15: Architektur von MoveIt! [26]

In Abbildung 15 sind die verschiedenen Bestandteile von MoveIt!, sowie deren Abhängigkeiten, abgebildet. Die Hauptbestandteile liegen in der Visualisierung, einer inversen Kinematik, einer Kollisionserkennung, Bahnplanungen, einer Roboteranbindung, Erfassung und Verarbeitung von Sensordaten und der Schnittstelle zu C++ und Python.

Über die *move\_group\_interface* Schnittstelle lässt sich z. B. der Roboter über Wegpunkte mithilfe von Posen programmieren. Darüber hinaus kann mit ihr eine kollisionsfreie Bahnplanung über Start- und Zielposen erstellt werden, siehe Abbildung 16.



**Abbildung 16:** Bahnplanung von Movelt!

Zur Roboteranbindung läuft auf der Robotersteuerung ein TCP-Nachrichtenserver [30] (mehr zur Netzwerkprogrammierung, siehe Abschnitt 2.5). Sobald eine Nachricht empfangen wurde, werden die Soll-Achswinkel aus dieser entnommen und direkt angefahren. Wenn die Robotersteuerung auch Multitasking beherrscht, werden in einem einstellbaren Takt die aktuellen Ist-Achswinkel zurückgesendet. Diese Roboterprogramme befinden sich für einige Industrieroboter verschiedener Hersteller in der ROS-Industrial Gruppe, unter anderem ABB, Fanuc und Motoman [39].

Da keines der vorgestellten Softwareprogramme und Bibliotheken die genauen Anforderungen erfüllt, wird in dieser Masterarbeit eine eigene Roboterbibliothek entwickelt. Diese benötigt ebenfalls eine inverse Kinematik, eine Kollisionserkennung und eine Visualisierung. Diese Komponenten könnten gegebenenfalls aus den Open Source Projekten entnommen werden.

## 4 Inverse Kinematik

Zur Steuerung der Bewegung eines Mehrkörpersystems im Raum ist die inverse Kinematik weit verbreitet. Durch sie werden die Achsstellungen für eine exakte Pose des Endeffektors berechnet. Verwendung findet sie neben der Robotik, auch in der Computeranimation [1, S. 1f].

Die Verfahren der inversen Kinematik lassen sich grob in drei Kategorien aufteilen:

- **Analytische Methoden**

Die analytischen Methoden berechnen die verschiedenen Konfigurationen der inversen Kinematik ohne Iteration und somit zuverlässig und schnell. Rein mathematische Lösungen können hingegen nicht gut mit Singularitäten (mehr zu Singularitäten, siehe Abschnitt 2.4.3) umgehen und liefern nicht immer die optimale Lösung.

Um dieses Verhalten zu verbessern, kann die analytische Methode mit einem numerisch stabilen, suchbasierten Ansatz verbessert werden, ohne die positiven Eigenschaften zu verlieren [4, S. 78].

Der Nachteil analytischer Methoden liegt im Umgang eines überbestimmten Systems. Als Lösung lassen sich einzelne Achsen aus der Berechnung entnehmen, solange es überbestimmt bleibt.

- **Numerische Methoden**

Diese Methoden basieren alle auf einem iterativem Prinzip. Es wird in nicht bestimmbarer Anzahl von Schritten versucht, die Pose des Endeffektors an die Zielpose anzunähern. Ein Problem der numerischen Methoden ist die robuste Berechnung schwer oder nicht erreichbarer Posen [1, S. 2]. Bei Methoden, die auf dem newtonschen Näherungsverfahren basieren, muss zudem die Ausgangskonfiguration und Schrittweite passend gewählt werden, um Divergenz zu vermeiden [6, S. 1]. Bedingt durch den iterativen Ansatz, ist die Rechenzeit höher als die der analytischen Methoden. Der Vorteil numerischer Methoden liegt darin, dass sie keine Probleme mit Singularitäten haben und meist direkt eine passende Lösung liefern. Zudem können sie mehr als sechs Achsen nutzbar machen und haben ebenfalls keine Probleme mit Mehrdeutigkeiten.

Viele numerische Methoden nutzen die Jacobi-Matrix um sich schrittweise an die gewünschte Pose anzunähern.

$$J(\theta) = \left( \frac{\delta s}{\delta \theta} \right) \quad (4.1)$$

Die Jacobi-Matrix aus Gleichung 4.1 ergibt sich aus der Endposenfunktion  $s$ , abgeleitet durch die Achswinkel  $\theta$ .

$$\Delta \vec{s} \approx J \Delta \theta \quad (4.2)$$

Ziel ist es nun, die Achswinkeländerung  $\Delta\theta$  so anzupassen, dass die Verschiebung der Pose  $\Delta \vec{s}$  genau auf die gewünschte Pose zeigt, siehe Gleichung 4.2 [1, S. 3-7].

Die folgenden verschiedenen Verfahren haben das Ziel, diese Achswinkeländerung  $\Delta\theta$  zu bestimmen:

Bei dem Jacobian-Transponse-Verfahren wird die Gleichung zur Bestimmung der Jacobi-Matrix stark vereinfacht. Statt der optimalen Inversen, wird die Transponierte der Matrix verwendet. Das Verfahren ist dadurch sehr simpel, aber auch sehr ungenau [1, S. 7].

Das Verfahren der Moore-Penrose-Inverse nutzt die Pseudoinverse der Jacobi-Matrix. Dieses Verfahren ist zwar genauer als das der Jacobian-Transponse, dafür aber in der Nähe von Singularitäten instabil [1, S. 8f].

Die Levenberg-Marquardt-Damped-Least-Squares-Methode basiert auf der Moore-Penrose-Inverse, ist aber numerisch stabil und hat keine Probleme mit Singularitäten. Statt den geringsten Fehler zur Sollpose zu verwenden, nutzt sie den geringsten geometrischen Abstand [1, S. 9f].

Mehr zur Damped-Least-Squares-Methode: [2]

- **Methoden der künstlichen Intelligenz**

Mithilfe von neuronalen Netzwerken lässt sich eine inverse Kinematik antrainieren. Dabei kann ein gewöhnliches, neuronales *feed-forward*-Netz verwendet werden. Die Trainingsdaten lassen sich schnell in großer Anzahl über die Vorwärtskinematik generieren. Vorteile der Methoden mit künstlicher Intelligenz liegen im guten Umgang mit Singularitäten, der Nutzbarkeit von mehr als sechs Achsen und der schnellen Berechnung. Nachteile bestehen in den hohen Ungenauigkeiten, den unvorhersehbaren Fehlern, den zufälligen Konfigurationsbestimmungen und der schlechten Wartbarkeit.

Mehr zu neuronalen Netzen: [10]

Mehr zur künstlichen Intelligenz für die inverse Kinematik: [5]

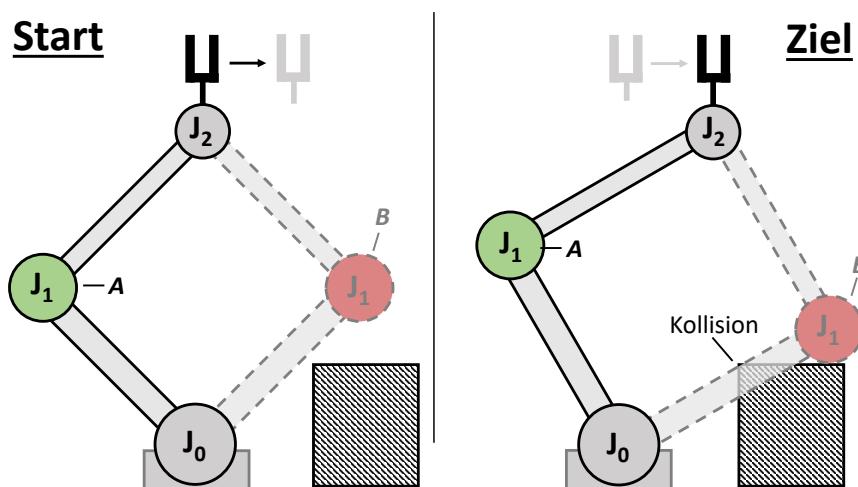
## 4.1 Methodenwahl

In der entwickelten Roboterbibliothek sollte die Umsetzung der inversen Kinematik robust sein, d. h. sie sollte ohne Probleme mit Singularitäten, Mehrdeutigkeiten und Erreichbarkeiten umgehen können. Die Bahnplanung, zusammen mit der Kollisionserkennung, setzt der inversen Kinematik voraus, dass sie ähnliche Konfigurationen für alle Posen einer vorgegebenen Bahn in geringer Schrittweite ermöglicht. Für diese Bahnen wird auch eine hinreichende Genauigkeit der Endeffektorpose benötigt. Eine möglichst schnelle Berechnung wäre wünschenswert, ist aber nicht nötig, da die Berechnung zeitunkritisch vor der Ausführung stattfinden kann.

Der Vorteil numerischer Methoden liegt darin, dass sie die nächste Pose ohne Konfigurationssprung finden, da sie durch die aktuelle Pose als Ausgangssituation schon in ihrer Nähe liegt. Somit wird die Startpose definiert und alle Zwischenpunkte auf der Bahn haben automatisch eine ähnliche Konfiguration, sofern sie mit dieser erreichbar sind.

Die Unbestimmtheit der Startkonfiguration wird aber zum Nachteil, wenn eine Pose auf dem Weg nicht mehr die passende Konfiguration aufweist. Dann muss diese Bahn abgebrochen werden, obwohl sie vielleicht mit einer anderen Startkonfiguration möglich wäre.

Abbildung 17 verdeutlicht dies noch einmal. Angenommen der Roboter befindet sich auf einer Bahn, die zum Start die Konfiguration  $B$  ohne Kollision eingenommen hat, wird die Bahn bei der Bahnplanung zum Ziel abbrechen. Dabei könnte die Bahn mit der richtigen Konfiguration, hier  $A$ , angefahren werden.



**Abbildung 17:** Vereinfachte Darstellung eines Konfigurationsfehlers bei einem Zweiachsen-Roboter

Um bei der Bahnplanung dementsprechend bessere Ergebnisse zu erzielen, werden für jeden Zwischenschritt alle möglichen Konfigurationen benötigt, um so eine passende Startkonfiguration für die gesamte Bahn auszuwählen.

Aus diesem Grund empfiehlt es sich, den „*minimal, numerically stable analytical inverse kinematics solver*“ aus OpenRAVE’s IKFast [4, S. 78ff] zu nutzen, da er für eine Pose direkt bis zu 16 mögliche Konfigurationen errechnet, keine Probleme mit Singularitäten hat und dabei schnell und numerisch stabil ist.

## 4.2 Analytisches Verfahren mit IKFast

Ein Nachteil numerischer Methoden ist die fehlende Möglichkeit der Berechnung mehrerer Konfigurationen zum Erreichen einer Pose. Dafür haben diese aber u. a. keine Probleme mit Singularitäten. Das Programm IKFast aus dem offenen Projekt OpenRAVE erzeugt roboterabhängigen Quellcode für eine Berechnung der inversen Kinematik, auf Grundlage analytischer Verfahren [4, S. 78].

Die Transformationskette zur Berechnung der Endeffektorpose ist die Basis analytischer Verfahren. Diese nutzt affine Transformationsmatrizen  $T_i$  der Achsabstände, zusammen mit den Matrizen der Achsstellungen  $J_i$ , siehe Gleichung 4.3 (mehr zur affinen Transformation, siehe Abschnitt 2.3).

$$T_{ee} = T_0 J_0 T_1 J_1 T_2 J_2 \dots T_n = \begin{bmatrix} r_{00} & r_{01} & r_{02} & p_x \\ r_{10} & r_{11} & r_{12} & p_y \\ r_{20} & r_{21} & r_{22} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.3)$$

In Gleichung 4.3 steht  $T_{ee}$  für die Pose des Endeffektors in Matrixdarstellung und  $J_i$  jeweils für eine  $4 \times 4$  Translations- bzw. Rotationsmatrix, abhängig von der Achsstellung. Mithilfe dieser Gleichung lässt sich dementsprechend die Vorwärtskinematik berechnen.

Durch Umstellen der Formel, siehe Gleichungen 4.4, lässt sich ein Gleichungssystem aufbauen, mit dem die inverse Kinematik auf analytischem Weg berechnet werden kann.

$$T_0^{-1}T_{ee}T_n^{-1} = J_0T_1J_1T_2J_2...J_{n-1}$$

$$J_0^{-1}T_0^{-1}T_{ee}T_n^{-1} = T_1J_1T_2J_2...J_{n-1}$$

$$T_1^{-1}J_0^{-1}T_0^{-1}T_{ee}T_n^{-1} = J_1T_2J_2...J_{n-1} \quad (4.4)$$

...

$$T_{n-1}^{-1}J_{n-1}^{-1}...T_0^{-1}T_{ee}T_n^{-1} = J_{n-1}$$

Das Umstellen des Gleichungssystems führt bei den normalen Varianten der analytischen Verfahren zu hochgradig univariaten Gleichungen, die bei Singularitäten schlecht konditioniert sind und so zu Problemen führen [4, S. 80f].

IKFast bedient sich ebenfalls an diesem Gleichungssystem, sucht aber zunächst nach Gleichungen geringerer Komplexität. Diese Komplexität ergibt sich entweder nach der Lösungskomplexität, welche die Anzahl an Lösungen einer Gleichung beschreibt, oder nach der numerischen Komplexität, welche aus der Anzahl an Rechenschritten zur Lösung besteht [4, S. 82-95].

Mithilfe von IKFast lässt sich die inverse Kinematik in vier verschiedenen Möglichkeiten erstellen [4, S. 79]:

- **Transformation IK (6DOF Translation+Rotation)**

Inverse Kinematik zum genauen Erreichen einer Pose mit dem Endeffektor. Diese Option erfüllt die Anforderungen der Bahnplanung dieser Arbeit.

- **Translation IK (3DOF)**

Inverse Kinematik zum genauen Erreichen einer Position, mit unbestimmter Orientierung.

- **Rotation IK (3DOF)**

Inverse Kinematik zum genauen Erreichen einer Orientierung, mit unbestimmter Position.

- **Look-at Ray IK (4DOF)**

Durch diese Option schaut der Endeffektor über eine Linie im Raum, dabei kann er sich auf der Linie frei bewegen und um die Linie rotieren.

Um das Programm zu starten, wird zunächst eine XML-Datei benötigt, welche die kinematische Kette des Roboters beinhaltet. Nach Abschluss der Berechnung wird eine unabhängige C++-Quellcodedatei erzeugt, die je nach Option eine der oben inversen Kinematiken löst. Mithilfe einer dazugehörigen C++-Headerdatei kann neben der inversen Kinematik auch die Vorwärtsskinematik berechnet werden.

### 4.3 Implementierung in Movelt!

Eine der wichtigsten Funktionen in Movelt! ist das *Pick and Place*. Dabei werden nur die Posen zum Greifen und Ablegen benötigt und je nach Anwendung noch eine kurze geradlinige Bewegung zum Anheben. Der Weg zwischen dem Greifen und Ablegen hat oft nur die Vorgabe, keine Kollision zu verursachen und das Bauteil nicht zu kippen. Für diese Anforderungen eignet sich die numerische inverse Kinematik, da sie mit Singularitäten gut zurecht kommt und die erstbeste Konfiguration zum Greifen bzw. Ablegen genügt. Für die inverse Kinematik nutzt Movelt! das numerische Verfahren KDL von Orocosp, welches auf dem Levenberg-Marquardt-Damped-Least-Squares beruht [34].

Innerhalb der Movelt! C++-Bibliothek gibt es eine Funktion, die nicht nur eine, sondern wenn möglich auch mehrere Konfigurationen für eine Pose errechnen kann. Die Standardimplementierung im Quellcode liefert hingegen immer nur eine Lösung, nämlich die der numerischen Methode [27, Zeile 161].

Es ist möglich, das analytische Verfahren von IKFast in Movelt! zu implementieren. Da hierbei auch mehrere Konfigurationen sofort berechnet werden, liefert die Movelt! Funktion dementsprechend auch mehrere Lösungen [28, Zeile 1259]. Das Problem von Movelt! ist, dass es diese Funktion selbst nicht nutzt, weil es standardmäßig auch nur eine Lösung erwartet.

Über die *move\_group\_interface* C++-Schnittstelle lässt sich eine Bahn programmieren, die der Roboter geradlinig zwischen den Wegpunkten abfährt. Umgesetzt wurde es, indem der Roboter zunächst über das numerische Verfahren zur Startposition gelangt und dabei eine unbestimmte Konfiguration erhält. Anschließend werden in einer bestimmbaren Schrittgröße, Zwischenschritte errechnet. In dieser Umsetzung hat das numerische Verfahren einen Vorteil, indem es durch die nahliegende Ausgangspose, direkt die passende Konfiguration für den nächsten Schritt wählt. Wenn es doch zu einer falschen Konfiguration kommt, wird die Bahn abgebrochen [29], wie bereits in Abschnitt 4.1 beschrieben.

Hier ist auch ein weiterer Nachteil in der Umsetzung in Movelt! erkennbar. Unabhängig davon, ob die numerische Variante oder IKFast genutzt wird, ist die Startkonfiguration unbestimmt und kann dazu führen, dass die Bahn abgebrochen wird, obwohl eine andere Konfiguration der Startpose eventuell funktioniert hätte.

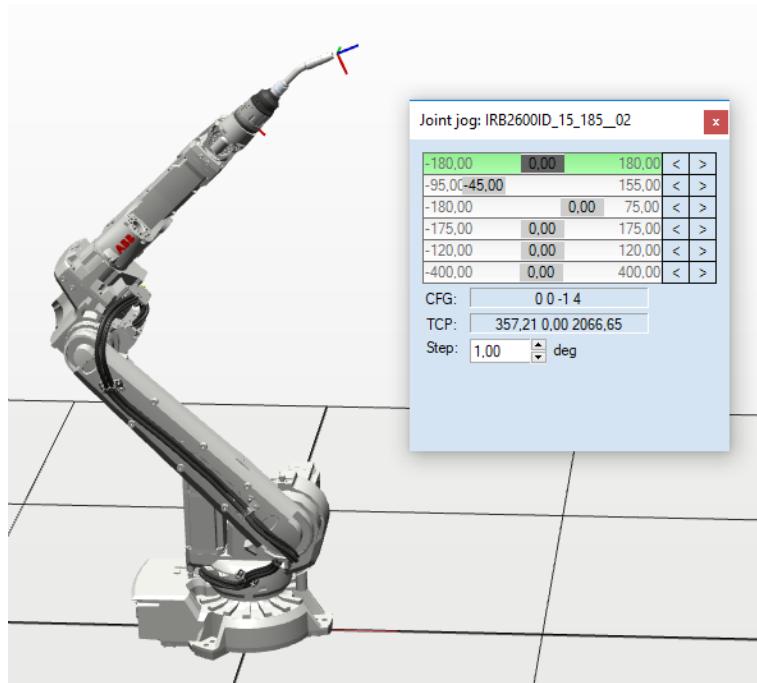
### 4.4 Abbildung der Roboterkinematik

Da Movelt! den Anforderungen der Bahnplanung dieser Masterarbeit nicht genügt, wird für die inverse Kinematik das Programm IKFast von OpenRAVE verwendet. Abgesehen von den Methoden der künstlichen Intelligenz, benötigen alle Verfahren die kinematische Kette des Roboters. Sie beschreibt die Transformationen einer

Achse zur Nächsten. Nach der Notation von Denavit und Hartenberg [3] zeigt die  $Z$ -Achse der Koordinatensysteme stets in Richtung der kinematischen Achse, sodass die Translation um  $Z$ , bzw. die Drehung um  $Z$ , die mechanische Achsbewegung darstellt (siehe Abschnitt 2.4.1).

#### 4.4.1 Ermittlung der kinematischen Kette

Da ABB nach Anfrage die kinematische Kette nicht zur Verfügung stellen konnte und sie auch nicht in den Dokumentationen zu finden ist, muss diese selbst eingemessen werden. Das hat darüber hinaus den Vorteil, dass die Fertigungstoleranzen an einem echten Roboter mit eingemessen werden könnten. Da zum Zeitpunkt des Verfassens dieser Arbeit der ABB IRB 2600ID Industrieroboter noch nicht vorhanden war, wurden die Transformationen mithilfe der Simulation gemessen, siehe Abbildung 18.



**Abbildung 18:** Messung der Achse 1 innerhalb RobotStudio

Bis auf die Roboterbasis und den Endeffektor, benötigen die Transformationsmatrizen nicht die genaue Position in Z-Richtung der Achse. Um eine Achse einzumessen, kann wie folgt vorgegangen werden:

Zuerst werden alle Achsen des Roboters wenn möglich auf null eingestellt. Dann wird nur die zu messende Achse verstellt, sodass drei Positionen  $A$ ,  $B$  und  $C$  des Endeffektor im Raum gemessen werden. Mithilfe dieser drei Positionen wird eine Transformationsmatrix  $T_O$  erstellt, die durch die drei Punkte im Raum aufgespannt wird, siehe Gleichungen 4.5 - 4.8.

$$\vec{ab} = \frac{B - A}{\|B - A\|_2} \quad (4.5)$$

$$\vec{az} = \frac{\vec{ab} \times (C - A)}{\|\vec{ab} \times (C - A)\|_2} \quad (4.6)$$

$$\vec{ac} = \vec{az} \times \vec{ab} \quad (4.7)$$

$$To = \begin{bmatrix} \vec{ab}_x & \vec{ac}_x & \vec{az}_x & A_x \\ \vec{ab}_y & \vec{ac}_y & \vec{az}_y & A_y \\ \vec{ab}_z & \vec{ac}_z & \vec{az}_z & A_z \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \quad (4.8)$$

$To$  beschreibt nun die Transformation in die, durch die drei Punkte  $A, B, C$  aufgespannte Fläche und somit die Orientierung der Rotationsachse.

$$b = To \cdot \begin{pmatrix} B \\ 1 \end{pmatrix} \quad (4.9)$$

$$c = To \cdot \begin{pmatrix} C \\ 1 \end{pmatrix} \quad (4.10)$$

Durch die Transformation der beiden Punkte  $B$  und  $C$ , siehe Gleichungen 4.9 und 4.10, kann anschließend, durch drei 2D-Punkte, der Mittelpunkt der Rotationsachse berechnet werden. Die Punkte  $b$  und  $c$  haben Koordinaten ausgehend vom Koordinatensystem der aufgespannten Fläche und somit einen Betrag von  $|z| = 0$ . Der dritte Punkt für die Kreismittelpunktberechnung liegt genau auf  $a_x = a_y = 0$ , da  $A$  der Ursprung des Koordinatensystems ist.

$$r^2 = (x - M_x)^2 + (y - M_y)^2 \quad (4.11)$$

Mithilfe der Gleichung 4.11 lässt sich als nächstes, durch Einsetzen der Punkte  $a, b$  und  $c$ , ein Gleichungssystem aufbauen, um den Kreismittelpunkt  $M$  zu bestimmen. Die Kombination zusammen mit der Transformationsmatrix  $To$ , ergibt die Transformationsmatrix  $T$ , von der Roboterbasis ausgehend zur Rotationsachse, siehe Gleichung 4.12.

$$T = T_o \cdot \begin{bmatrix} 1 & 0 & 0 & -M_x \\ 0 & 1 & 0 & -M_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.12)$$

Da viele Industrieroboter die Rotationsachsen jeweils um  $90^\circ$  versetzt angeordnet haben, genügt es in der Simulation bereits, direkt den Achsenmittelpunkt aus den drei 2D-Punkten zu bestimmen. Die Transformationsmatrizen lassen sich dann hän-disch mit  $90^\circ$  Rotationen aufbauen.

Die folgenden Transformationsmatrizen beschreiben die Kinematik des für den For-schungsprojekt genutzten Industrieroboters.

Achse 0	Achse 1	Achse 2
$\begin{bmatrix} 1 & 0 & 0 & 0,000 \\ 0 & 1 & 0 & 0,000 \\ 0 & 0 & 1 & 0,000 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0,150 \\ 0 & 0 & -1 & 0,000 \\ 0 & 1 & 0 & 0,445 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0,150 \\ 0 & 0 & -1 & 0,000 \\ 0 & 1 & 0 & 1,345 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Achse 3	Achse 4	Achse 5
$\begin{bmatrix} 0 & 0 & -1 & 0,000 \\ 0 & 1 & 0 & 0,000 \\ 1 & 0 & 0 & 1,495 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0,936 \\ 0 & 0 & -1 & 0,000 \\ 0 & 1 & 0 & 1,495 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & -1 & 1,071 \\ 0 & 1 & 0 & 0,000 \\ 1 & 0 & 0 & 1,495 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Schweißwerkzeug		
$\begin{bmatrix} 0,927 & 0,000 & -0,375 & 0,050 \\ 0,000 & 1,000 & 0,000 & 0,000 \\ 0,375 & 0,000 & 0,927 & 0,372 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		

#### 4.4.2 Ausführung von IKFast

OpenRAVE nutzt zur Beschreibung der Roboter und dessen Kinematik, Dateien im XML-Format. In diesen lassen sich Gruppen kinematischer Körper und Roboter beschreiben. Dabei besteht ein Roboter (`robot`) aus einem kinematischen Körper (`kinbody`), welcher wiederum aus einzelnen Körpern (`body`) und Achsen (`joint`) be-steht. Innerhalb eines Körpers werden Geometrien entweder mit geometrischen Pri-mitiven, wie Zylinder, oder über CAD-Dateien beschrieben. Außerdem wird die Po-se des Körpers im Raum und die Pose des Geometrieursprungs definiert. Über die Achsen wird beschrieben, zwischen welchen Körpern, an welcher Pose, die Dreh-achsen vorkommen.

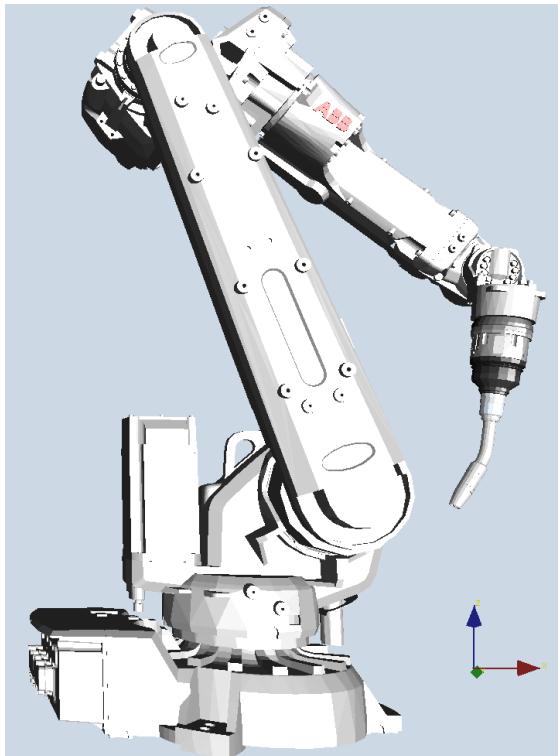
Die Posen der Körper werden in Translation und Rotation dargestellt. Für die Beschreibung der Rotation gibt es die drei Möglichkeiten:  
Quaternionen (quat), Rotationsachsen (rotationaxis)  
und Rotationsmatrizen (rotationmat) (mehr zu Posen, siehe Abschnitt 2.2).

Mehr zum OpenRAVE XML-Format, siehe [33]

Mithilfe der gemessenen Transformationsmatrizen, können nun die Körperposen beschrieben werden. Da die Koordinatensysteme der CAD-Dateien von ABB Industrierobotern stets in der Roboterbasis liegen, müssen diese für die Geometrie der Körper wieder zurück zur Roboterbasis transformiert werden. Hierzu werden für die Posen der Geometrie jeweils die Inversen der gemessenen Transformationsmatrizen verwendet.

Das Werkzeug des Roboters wird über einen Manipulator (`manipulator`) innerhalb der Robotergruppe beschrieben. Er kann ebenfalls Geometrien beinhalten und wird für die Beschreibung der Endeffektorpose, also dem TCP, verwendet.

Die fertigen XML-Dateien (siehe Quellcode 10 - 13 in Anhang A) können mithilfe der OpenRAVE Visualisierung angezeigt und überprüft werden, siehe Abbildung 19.



**Abbildung 19:** ABB IRB 2600ID in der Visualisierung von OpenRAVE, mit den Achsstellungen (0°, -20°, 60°, 0°, 60°, 0°)

Das Programm IKFast nutzt nun diese Roboterbeschreibung zur Generierung einer C++-Quellcodedatei, welche die Vorwärtskinematik und inverse Kinematik beinhaltet. Beim Starten des Programms wurde die Option Transformation IK (6DOF Translation+Rotation) eingestellt.

Der generierte Quellcode kann nun über einige Schnittstellenfunktionen angesprochen werden. Die wichtigsten beiden sind:

- **ComputeFk**

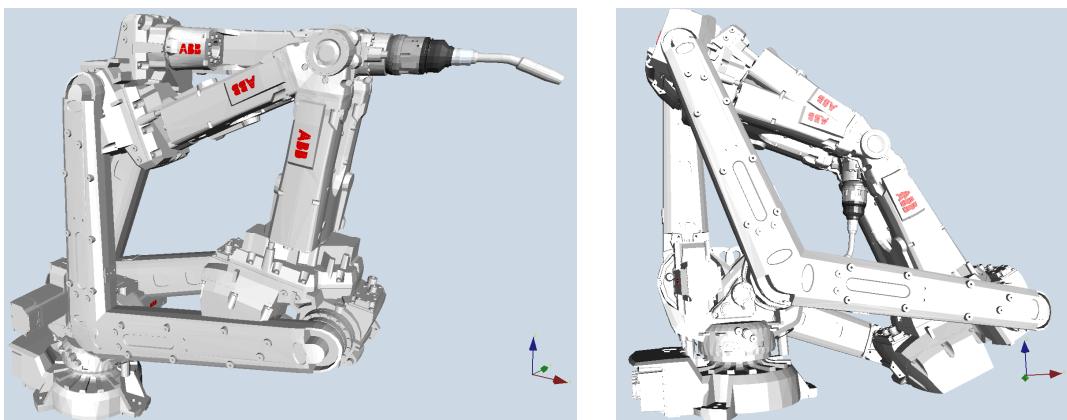
Diese Funktion berechnet die Vorwärtskinematik. Es werden die Achswinkel einer Konfiguration übergeben, um die dazugehörige Endeffektorpose zu berechnen. Die Schnittstellen nutzen zur Beschreibung der Pose eine affine Transformationsmatrix.

- **Computelk**

Mit dieser Funktion wird die inverse Kinematik berechnet. Hier wird eine Pose übergeben und eine Liste von maximal 16 Lösungen berechnet. Eine Lösung beschreibt neben der Konfiguration, ob und welche Achse in dieser Pose eine Singularität erzeugt. Solche Achsen werden in der Lösung als *free joint* beschrieben. Es ist möglich die inverse Kinematik nochmal zu berechnen und diese Achsen vorher selbst zu definieren.

#### 4.4.3 Auswertung des erzeugten Quellcodes

Die inverse Kinematik aus IKFast für den ABB IRB 2600ID ermittelt für viele Posen bis zu sieben Konfigurationen. Abbildung 20a zeigt die Konfigurationen, welche sich aus der Standardpose ermitteln lassen (Pose aus der Konfiguration:  $0^\circ, 0^\circ, 0^\circ, -0^\circ, 0^\circ, 0^\circ$ ). Abbildung 20b zeigt alle Konfigurationen für die Pose aus Abbildung 19 (Pose aus der Konfiguration:  $0^\circ, -20^\circ, 60^\circ, 0^\circ, 60^\circ, 0^\circ$ ).



**Abbildung 20:** Alle Lösungen der inversen Kinematik in der OpenRAVE-Visualisierung

Wie in den Abbildungen erkennbar, werden die Achsbegrenzungen bei der Berechnung der inversen Kinematik nicht mit einbezogen. In den XML-Dateien zur Roboterbeschreibung von OpenRAVE lassen sich zwar bei den Achsen (`joint`) Grenzen einstellen, diese werden aber bisher von IKFast ignoriert.

Somit müssen die ermittelten Konfigurationen, bei denen bereits eine Achse außerhalb ihrer Begrenzung liegt, aussortiert werden.

Weiterhin berechnet IKFast nur Konfigurationen mit Achsstellungen zwischen  $-\pi$  und  $\pi$ , bzw.  $-180^\circ$  und  $180^\circ$ . Sobald die Achsbegrenzung einer Achse des Roboters hinter dieser Begrenzung liegt, gibt es eventuell noch weitere Lösungen. Hierfür werden neue Konfigurationen aus den bisherigen erzeugt, indem an dieser Achse mehrmals  $\pm 2\pi$  aufaddiert wird. Das wird solange wiederholt, bis eine neue Konfiguration außerhalb der Achsbegrenzung liegt. Wie in Tabelle 1 zu erkennen ist, liegt die Achsbegrenzung der Achse 5 des ABB IRB 2600ID über  $\pm\pi$ , bzw.  $\pm 180^\circ$ .

Ein weiteres Problem ist ein Ergebnis der inversen Kinematik von genau  $\pm\pi$ , bzw.  $\pm 180^\circ$ . Angenommen es wird ein Achswinkel von  $+180^\circ$  errechnet. Wenn dessen Achsbegrenzung bei  $-180^\circ$  bis  $+75^\circ$  liegt, ist die Konfiguration möglich, wenn anstelle von  $+180^\circ$ ,  $-180^\circ$  verwendet wird.

**Tabelle 1:** Achsbegrenzungen des ABB IRB 2600ID

Achse	Von	Bis
0	$-180^\circ$	$180^\circ$
1	$-95^\circ$	$155^\circ$
2	$-180^\circ$	$75^\circ$
3	$-175^\circ$	$175^\circ$
4	$-120^\circ$	$120^\circ$
5	$-400^\circ$	$400^\circ$

Mithilfe eines Testprogramms (siehe Quellcode 14 in Anhang B) wurde die mit IKFast generierte inverse Kinematik des ABB IRB 2600ID Industrieroboters getestet. Tabelle 2 zeigt die Ergebnisse des Testprogramms. In dem Programm wurde der Fehler der Achsbegrenzungen zunächst ignoriert. Der Test lief dabei für eine Berechnung wie folgt ab:

Zunächst wurde eine zufällige Konfiguration erstellt, mit Achswinkeln zwischen  $-\pi$  und  $\pi$ . Mithilfe der Vorwärtskinematik wurde dann die dazugehörige Pose bestimmt. Somit ist eine Pose bekannt, die definitiv mindestens eine Lösung besitzt. Aus dieser Pose wurde anschließend wieder rückwärts die inverse Kinematik berechnet und getestet.

**Tabelle 2:** IKFast Ergebnisse zufälliger Konfigurationen des ABB IRB 2600ID

		Lösungen	Anzahl
Anzahl Berechnungen	100.000.000	Eine	8
Gescheiterte inverse Kinematik	78	Zwei	8
Fehlerhafte Konfiguration	0	Drei	16275604
Gefundene Singularitäten	90	Vier	0
Gemittelte Dauer der inverse Kinematik Berechnung	12,306 ms	Fünf	31
		Sechs	22
		Sieben	83724249

Wie die Ergebnisse in Tabelle 2 zeigen, konnten die Konfigurationen von 99,999922% aller Posen bestimmt werden. IKFast findet Singularitäten nur, wenn sie genau auf dieser Konfiguration liegen. Da eine kleine Abweichung (z. B. eine Winkelminute) bereits ausreicht, hat dieser Test nur sehr wenige gefunden.

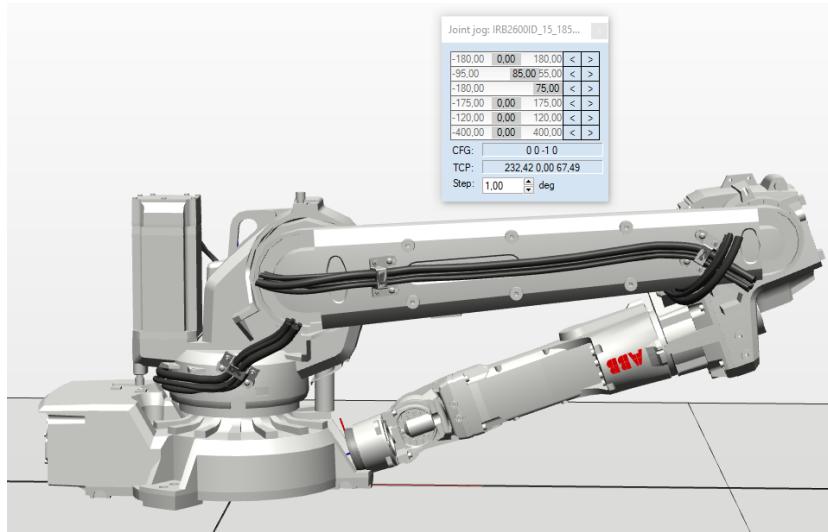
## 5 Kollisionserkennung

Nachdem die inverse Kinematik, Konfigurationen des Industrieroboters für eine Pose bestimmt hat, sollten diese auch auf Kollisionen überprüft und gegebenenfalls aussortiert werden. Kollisionen und Abstände zwischen Geometrien können bereits mehrere freie Bibliotheken berechnen. ODE [31] oder Bullet [18] werden für physikalische Dynamiksimulationen verwendet und beinhalten Module zur Kollisionserkennung.

Um die Kollision des Industrieroboters bei einer Bewegung von Pose *A* nach Pose *B* zu überprüfen, werden viele Posen in einer beliebigen Schrittweite von z. B. 1 mm auf der Bahn erstellt.

Alle diese Posen werden nun mit der passenden Konfiguration aus der inversen Kinematik auf Kollision getestet.

Da der für das Forschungsprojekt vorgesehene Industrieroboter ABB IRB 2600ID mit sich selbst kollidieren kann, siehe Abbildung 21, muss der Roboter zur Kollisionskontrolle zum einen mit sich selbst und zum anderen mit der Umgebung auf Kollision getestet werden.

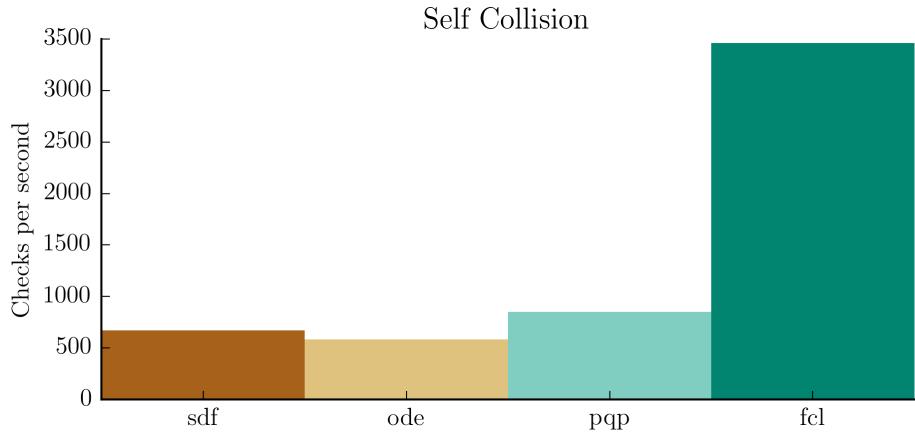


**Abbildung 21:** ABB IRB 2600ID in RobotStudio, mit einer Selbstkollision

### 5.1 Flexible Collision Library

FCL [23] ist eine C++-Bibliothek, die drei Arten der Kollisionsberechnung beherrscht. Neben der reinen Kollisionserkennung und der Abstandsmessung, kann sie auch Kollisionen mit Toleranzen testen. Für die Geometrie der Modelle werden unterschiedliche Darstellungen unterstützt, wie Quader, Zylinder, Kugeln und Flächen. Weiterhin werden auch konvexe und triangulierte Körper, wie CAD-Dateien unterstützt.

Verglichen mit anderen Bibliotheken zur Kollisionserkennung, ist die Berechnung in FCL schneller, siehe Abbildung 22.



**Abbildung 22:** Vergleich verschiedener Kollisionserkennungen [24]

Mithilfe eines Plugins [24], lässt sich FCL auch in OpenRAVE nutzen.

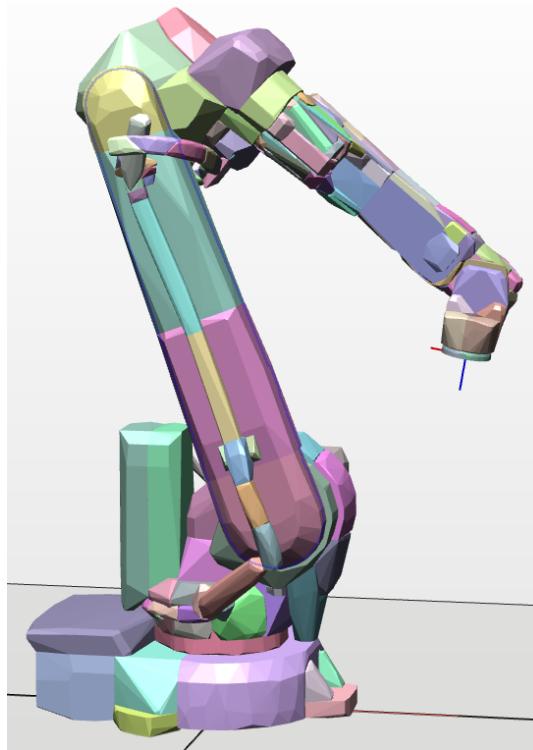
## 5.2 CAD-Dateien zur Kollisionserkennung

Zur Visualisierung der Bahnplanung können die CAD-Dateien der Roboterachsen angezeigt werden. Für die Kollisionserkennung sollte hingegen ein Sicherheitsabstand mit einbezogen werden. Entweder wird dieser bei der Berechnung der Kollisionserkennung berücksichtigt, oder das Kollisionsmodell der Achsen wird vergrößert.

## 5.3 Konvexe Zerlegung

Im Vergleich zu komplexen triangulierten Körpern, wie CAD-Dateien, sind konvexe Körper in mathematischen Berechnungen, wie der Kollisionserkennung, viel effektiver [9, S. 121]. Die einfache konvexe Hülle der Roboterachsen führt aber zu ungenauen Modellen.

Mithilfe der konvexen Zerlegung, wird ein komplexer Körper in mehrere konvexe Körper zerlegt. Die Berechnung der einzelnen konvexen Körper beschleunigt die Berechnung der Kollisionserkennung, ohne das Modell signifikant zu vereinfachen, siehe Abbildung 23.



**Abbildung 23:** ABB IRB 2600ID in RobotStudio, mit konvexer Zerlegung der Achsen zur Kollisionserkennung

Neben ABB, unterstützt auch OpenRAVE die konvexe Zerlegung.

In der folgenden Tabelle 3 wurde die durchschnittliche Zeit der Kollisionsüberprüfung mit und ohne Abstandsmessung gemessen. Um eine Toleranz auf die CAD-Dateien der Geometrie zu beaufschlagen, kann u. a. die Abstandsmessung genutzt werden. Sobald der Abstand kleiner als die Toleranz ist, wird eine Kollision gemeldet.

**Tabelle 3:** Durchschnittliche Rechenzeiten der Kollisionserkennung

Verfahren	Zeit pro Test
Reiner Kollisionstest	0,155 ms
Mit Abstandsmessung	1464,385 ms

Da eine Kollisionsüberprüfung mit Abstandsmessung deutlich länger dauert, wird auf diese verzichtet. Stattdessen sollte ein Kollisionsmodell generiert werden, welches bereits die Toleranz beinhaltet. Das hat zudem den Vorteil, dass verschiedene Achsen, verschiedene Toleranzen besitzen können.

Während des Verfassens dieser Masterarbeit wird aus Zeitgründen auf das Erstellen solcher Kollisionsgeometrien verzichtet. Zum Testen der entwickelten Roboterbibliothek genügt als Kollisionsmodell das CAD-Modell des Roboters.

## 6 Bahnplanung

Der elementare Kern dieser Roboterbibliothek besteht in der Planung von Roboterbahnen. Wie in Kapitel 3 vorgestellt, spezialisieren sich viele Softwareprogramme und Bibliotheken auf das *Pick and Place* und der vordefinierten Roboterprogrammierung. Für das *Pick and Place* genügt bei der Operation, die erstbeste Konfiguration des Roboters zum Greifen und Verfahren. Ist hingegen eine definierte Bahn notwendig, wie eine Schweißbahn, kann die erstbeste Startkonfiguration auf dem Weg zu einer Kollision führen oder nicht mehr in ähnlicher Konfiguration erreichbar sein, siehe Abschnitt 4.1 Abbildung 17.

Die hier vorgestellte Bahnplanung nutzt Posen für die gewünschten Roboterbahnen und bestimmt mithilfe der inversen Kinematik und der Kollisionserkennung, die notwendigen Konfigurationen des Roboters, falls die Bahnen möglich sind. Weiterhin kümmert sich die Bahnplanung um den kollisionsfreien Weg zwischen zwei Konfigurationen, um die errechneten Startkonfigurationen der Bahnen erreichen zu können.

### 6.1 Bahnbeschreibung

Zur Manipulation des Endeffektors im Raum, wird die inverse Kinematik des Roboters verwendet. Hierdurch kann die Position und Orientierung, bzw. die Pose des Endeffektors zur Beschreibung genutzt werden. Im Beispiel Schweißroboter werden häufig lineare Bewegungen des Schweißwerkzeugs benötigt. Um dies zu ermöglichen, wird diese Bahn durch mehrere Posen beschrieben, die das Werkzeug auf einem bestimmten Weg abfahren soll. In dieser Arbeit wurden zunächst lineare und zirkulare Wege zwischen den Posen realisiert. Eine Bahn besteht somit aus mehreren Wegpunkten, die auf verschiedenen Arten angefahren werden.

In der üblichen, vorherbestimmten, Roboterprogrammierung, wird nicht zwischen Bearbeitungsbahnen und den Zwischenwegen zum Erreichen dieser unterschieden. Der Roboter wird so programmiert, dass er auf einem bestimmten Weg die entsprechende Bahn erreicht, dann wird gegebenenfalls das Werkzeug aktiviert und die nachfolgende Bahn bearbeitet.

Für die unbestimmte Roboterprogrammierung ist es sinnvoll, zwischen den eigentlichen Prozessbahnen und den Zwischenwegen zu unterscheiden. Da die Bahnen stets anders im Raum vorliegen können, lassen sich die benötigten Konfigurationen des Roboters schwerer vorherbestimmen. Werden am Beispiel des Schweißroboters, mehrere Schweißnähte in mehrere Bahnen getrennt, können komplett unterschiedliche Konfigurationen bestimmt werden, die nicht mehr voneinander abhängig sind. Wie der Roboter den Start einer Bahn anfährt, ist oft irrelevant, sofern er auf dem Weg keine Kollision verursacht.

Um die passenden kollisionsfreien Konfigurationen einer Bahn berechnen zu können, kann wie folgt vorgegangen werden:

Zunächst werden alle möglichen Konfigurationen der Startpose des ersten Wegpunktes der Bahn bestimmt. Mithilfe der inversen Kinematik IKFast von OpenRAVE werden alle Konfigurationen zum Erreichen der Pose ermittelt. Diese werden nachträglich mithilfe der Achsbegrenzungen aussortiert. Falls eine Achse die Achsbegrenzung  $\pm\pi$ , bzw.  $\pm180^\circ$  überschreitet, werden eventuell noch weitere Konfigurationen ergänzt, siehe Abschnitt 4.4.3.

Alle übrigen Konfigurationen werden nachträglich auf Kollision getestet und gegebenenfalls aussortiert.

In der hier entwickelten Roboterbibliothek gibt es ein C++-Modul für die inverse Kinematik. Dieses beinhaltet u. a. Funktionen zum Einstellen der Achsbegrenzungen und den maximalen Achsgeschwindigkeiten.

- **getConfigurations**

Diese Funktion berechnet alle möglichen kollisionsfreien Konfigurationen, für eine übergebene Pose.

Um einen Wegpunkt der Bahn zu überprüfen, werden in einer einstellbaren Schrittweite, von z. B. 1 mm, Zwischenposen generiert. Von einer Zwischenpose zur nächsten, darf der Roboter seine Konfiguration nur leicht anpassen, da er sonst bei falscher Konfiguration einen großen Weg mit seinen Achsen verfährt.

Zur Berechnung einer naheliegenden Konfiguration, besitzt das Modul der inversen Kinematik zwei Funktionen, welche jeweils bei Erfolg die errechnete Konfiguration zurückgeben und eine Pose mit der naheliegenden Konfiguration annimmt:

- **getNearConfiguration**

Diese Funktion berechnet für die übergebene Pose zunächst alle möglichen Konfigurationen. Als weiteren Parameter lässt sich die maximale Achswinkeländerung einer Achse zwischen der alten und neuen Konfiguration einstellen.

$$\Delta\theta_{max} = ||config_{near} - config_{new}||_\infty \quad (6.1)$$

Gleichung 6.1 zeigt, wie die maximale Achswinkeländerung  $\Delta\theta_{max}$  aller Achsen von der alten nahen Konfiguration  $config_{near}$ , zur neuen Konfiguration  $config_{new}$  berechnet werden kann (mehr zur  $\infty$ -Norm, siehe Abschnitt 2.1 Gleichung 2.5).

Um Rechenzeit zu sparen, wird eine passende Konfiguration erst im Anschluss auf Kollision geprüft und bei Erfolg, die mit der geringsten Abweichung zurückgegeben.

- **getNearConfigurationWithSpeed**

Da viele Prozesse eine bestimmte Endeffektorgeschwindigkeit benötigen, dürfen die maximalen Achsgeschwindigkeiten der einzelnen Achsen auf dem Weg von der alten, zur neuen Konfiguration nicht überschritten werden. Diese Funktion nimmt als zusätzlichen Parameter die Zeit, die von der alten, zur neuen Konfiguration vergeht. Diese Zeit lässt sich durch die Endeffektorgeschwindigkeit zusammen mit der Schrittweite bestimmen.

$$toofast = \sum_{j=0}^5 \frac{|config_{nearj} - config_{newj}|}{time} > maxjointspeeds_j \quad (6.2)$$

$$\Delta J_{sum} = ||config_{near} - config_{new}||_1 \quad (6.3)$$

Mithilfe von Gleichung 6.2 kann bestimmt werden, ob die maximalen Achsgeschwindigkeiten eingehalten wurden. Dabei wird für jede Achse  $j$  überprüft, ob ihre benötigte Achsgeschwindigkeit größer als die passende maximale ist. Für den Fall, dass mehrere Lösungen vorhanden sind, wird mit Gleichung 6.3 die Lösung zurückgegeben, die den kleinsten Weg aller Achsstellungen zurücklegen muss (mehr zur *Summennorm*, siehe Abschnitt 2.1 Gleichung 2.4).

Tabelle 4 zeigt die maximalen Achsgeschwindigkeiten des Industrieroboters, der für das Forschungsprojekt genutzt wird.

**Tabelle 4:** Maximale Achsgeschwindigkeiten des ABB IRB 2600ID

Achse	Maximalgeschwindigkeit
0	$\pm 175^\circ/s$
1	$\pm 175^\circ/s$
2	$\pm 175^\circ/s$
3	$\pm 360^\circ/s$
4	$\pm 360^\circ/s$
5	$\pm 500^\circ/s$

Nach Bestimmung der Startpose einer Bahn, wird nun mit einer einstellbaren Schrittweite und einer der oberen zwei Funktionen die gesamte Bahn auf Kollision und Erreichbarkeit überprüft. Falls an einer Stelle eine Konfiguration nicht ermittelt werden konnte, wird nochmal von vorne mit der nächsten Startkonfiguration getestet.

Damit die Bahnplanung keine zufällige Reihenfolge der Startkonfigurationen nutzt, werden die möglichen Konfigurationen der Startpose sortiert. Im Bibliotheksmodul der inversen Kinematik gibt es hierfür folgende Funktion:

- **sortByVotes**

Als Parameter nimmt diese Funktion eine Liste der Konfigurationslösungen an und sortiert diese. Dazu enthält das Modul eine Standardkonfiguration *default* und einen Aufwandsfaktor für jede Achse *votes*.

$$v = \|(config - default) \odot votes\|_1 \quad (6.4)$$

Gleichung 6.4 errechnet den Aufwand  $v$  einer Konfiguration, indem es den Achsabstand jeder Achse zur Standardkonfiguration berechnet und bewertet. Die elementweise Multiplikation  $\odot$  errechnet hierbei für jede Achsänderung einen achsabhängigen Aufwandsfaktor auf. Somit lässt sich den ersten Achsen ein höherer Aufwand zuweisen, als der letzten. Mithilfe der Konfigurationsaufwände lässt sich nun die Liste der Konfigurationen sortieren.

**Tabelle 5:** Beispiel einer Standardkonfiguration und der Achsaufwände

Achse	Standardwinkel	Aufwand
0	$0^\circ$	1.0
1	$0^\circ$	1.0
2	$0^\circ$	1.0
3	$0^\circ$	0.5
4	$0^\circ$	0.5
5	$0^\circ$	0.1

Tabelle 5 zeigt die bei der Entwicklung genutzten Standardwinkel und Aufwände. Bei der Konfiguration des Roboters mit allen Achsen  $0^\circ$  befindet sich der Roboter, oberhalb eines Arbeitstisches, in einer Standardlage für diese Umgebung. Die Aufwände wurden so gewählt, dass der Einfluss der ersten drei Achsen höher liegt, als die der letzten. Beide Einstellungen hängen später individuell vom Prozess ab und sollten dementsprechend geschätzt werden.

In der entwickelten Roboterbibliothek hat jeder Wegpunkt den selben Grundaufbau:

- **start**

Die Pose, die den Start des Wegpunktes beschreibt.

- **goal**

Die Pose am Ziel des Wegpunktes.

- **length**

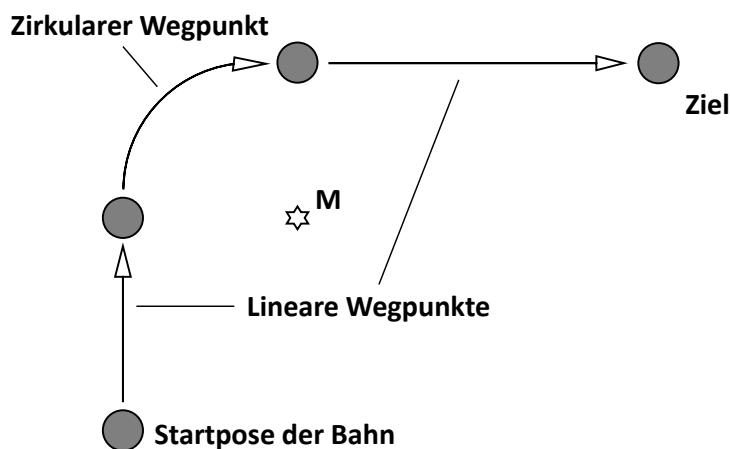
Diese Funktion liefert nach Aufruf die Länge des Weges in *m* zurück.

- **sample**

Mithilfe dieser Funktion lassen sich Zwischenposen des Wegpunktes berechnen. Dazu wird mit dem Parameter *t* (in %) angegeben, wo die Pose auf dem Wegpunkt liegt.

Im Modul Wegpunkte wurden bereits zwei Arten implementiert, nämlich lineare und zirkuläre. Abbildung 24 zeigt den schematischen Aufbau einer Bahn, bestehend aus zwei linearen und einem zirkularem Wegpunkt. Die Startpose des folgenden Wegpunktes einer Bahn, ist immer die Zielpose des vorherigen.

Mithilfe der Funktion `length`, lässt sich nun, unabhängig von der Art, die Länge des Weges bestimmen. Diese und die Schrittweite ergeben die Anzahl der Zwischenposen. Welche dann anschließend, ebenfalls unabhängig von der Art, durch die Funktion `sample` generiert werden.



**Abbildung 24:** Schema einer Bahn, bestehend aus drei Wegpunkten

### 6.1.1 Lineare Wegpunkte

Zur Implementierung einer Wegpunktart, gehört die Erstellung der Funktionen `length` und `sample`.

Für lineare Wege ist die Länge der geometrische Abstand zwischen der Start- und Zielpose, siehe Gleichung 6.5.

$$\text{length} = \|\text{position}_{\text{goal}} - \text{position}_{\text{start}}\|_2 \quad (6.5)$$

Für die `sample` Funktion müssen die Zwischenposen interpoliert werden. Für die Position genügt die lineare Interpolation, siehe Gleichung 6.6, wobei  $t$  die gewünschte Zwischenpose (in %) beschreibt.

$$\text{position}_{\text{sample}} = (\text{position}_{\text{goal}} - \text{position}_{\text{start}}) \cdot t + \text{position}_{\text{start}} \quad (6.6)$$

Die Interpolation der Orientierung hingegen, gestaltet sich nicht so trivial [12, S. 1]. Wird lediglich die Orientierung als Eulerwinkel linear interpoliert, rotiert die Pose nicht auf dem kürzestem Weg. Das liegt daran, dass die Eulerwinkel nacheinander transformiert werden und somit gegenseitig abhängig sind. Auch bei linearer Interpolation der Rotationsachsen, Quaternionen und Rotationsmatrizen, wird nicht in kürzestem Weg interpoliert [12, S. 2ff] (mehr zu den verschiedenen Darstellungen einer Orientierung, siehe Abschnitt 2.2).

Mithilfe des *SLERP*-Algorithmus (*spherical linear interpolation*) lässt sich die Orientierung mit gleichbleibender effektiver Winkelgeschwindigkeit realisieren.

Der *SLERP*-Algorithmus nutzt hierzu Quaternionen als Orientierungsdarstellung.

Die lineare Algebra C++-Bibliothek Eigen [22] besitzt bereits eine Implementierung des *SLERP*-Algorithmus.

Mehr zum *SLERP*-Algorithmus: [12, S. 3-7]

### 6.1.2 Zirkuläre Wegpunkte

Eine Kreisfahrt im Raum lässt sich durch die Pose des Mittelpunktes zusammen mit dem Radius beschreiben. Die  $X$ -Achse der Mittelpunktpose zeigt dabei in Richtung der Startpose. Die  $Y$ -Achse der Mittelpunktpose zeigt in die Richtung des Kreisbogens und somit bei einem Kreisbogen von genau  $\pi/2$ , bzw.  $90^\circ$ , auf die Zielpose.

$$\text{position}_{\text{sample}} = n_x \cdot \cos(\alpha \cdot t) \cdot r + n_y \cdot \sin(\alpha \cdot t) \cdot r + M \quad (6.7)$$

Gleichung 6.7 zeigt die Formel um die Positionen der Zwischenpunkte zu berechnen. Hierbei ist  $r$  der Radius,  $M$  die Position der Mittelpunktpose und  $n_x$ , bzw.  $n_y$  sind die normierten Vektoren der  $X$ - und  $Y$ -Achse der Mittelpunktpose.  $\alpha$  ist der Winkel des Kreisbogens von der Start-, zur Zielpose.

Um die `sample` Funktion zu implementieren, fehlt noch die Interpolation der Orientierung. Diese kann genauso wie bei der linearen Interpolation mit dem *SLERP*-Algorithmus umgesetzt werden. Da es allerdings bei einem Kreisbogen von  $\geq \pi$ , bzw.  $\geq 180^\circ$ , nicht eindeutig ist, in welche Richtung interpoliert werden muss, ist eine Zwischenpose notwendig.

$$\text{length} = \alpha \cdot r \quad (6.8)$$

Gleichung 6.8 zeigt die Berechnung für `length` und beschreibt den Umfang des Kreisbogens. Hierbei sollte  $\alpha$  in Radian angegeben werden.

Für den zirkularen Wegpunkt werden zur Erstellung mehrere Parameter benötigt. Da es aufwendig ist, die Mittelpunktpose, den Radius und den Winkel des Kreisbogens, zur Anwendung zu bestimmen, ist eine vereinfachte Variante vorhanden. Diese ermittelt alle zusätzlichen Parameter aus einer weiteren Pose. Zur Erstellung wird somit die Start-, Ziel- und Zwischenpose benötigt. Letztere sollte mit der richtigen Orientierung auf dem Kreisbogen liegen und darf maximal  $< \pi$ , bzw.  $< 180^\circ$ , von der Start- und Zielpose entfernt sein. Die noch benötigten Parameter werden wie folgt ermittelt:

Zunächst wird der Mittelpunkt des Kreisbogens über die drei vorhandenen Positionen der Posen ermittelt:  $\text{position}_{start}$ ,  $\text{position}_{goal}$  und  $\text{position}_{via}$ . Für die Berechnung des Mittelpunktes  $M$ , kann wie in Abschnitt 4.4.1 vorgegangen werden.

$$r = \|\text{position}_{start} - M\|_2 \quad (6.9)$$

$$n_x = \frac{\text{position}_{start} - M}{r} \quad (6.10)$$

$$n_z = \frac{n_x \times (\text{position}_{via} - M)}{\|n_x \times (\text{position}_{via} - M)\|_2} \quad (6.11)$$

$$n_y = n_z \times n_x \quad (6.12)$$

Die Gleichungen 6.9 - 6.12 zeigen die Berechnungen des Radius  $r$  und der benötigten normierten Achsen der Mittelpunktpose  $n_x$  und  $n_y$ . Durch den Vektor, der vom Mittelpunkt  $M$  zur Position der Zwischenpose  $position_{via}$  zeigt, lässt sich mit dem Kreuzprodukt des Vektors  $n_x$ , die Rotationsachse  $n_z$  bestimmen. Diese ist notwendig um den normierten Vektor  $n_y$  zu bestimmen, der zum Kreisbogens hin gerichtet sein muss.

Für den Winkel  $\alpha$  des Kreisbogens kann das Skalarprodukt, aus  $n_x$  und dem normierten Vektor von der Mittelpunktpose zur Zielpose, nicht verwendet werden. Das liegt daran, dass das Skalarprodukt nur zum Berechnen des kleinsten Winkels der beiden Vektoren genutzt werden kann, in diesem Fall aber ein Winkel von  $> \pi$ , bzw.  $> 180^\circ$ , vorkommen kann.

$$trans_{goal} = M^{-1} \cdot position_{goal} \quad (6.13)$$

$$\alpha = atan2(trans_{goal\ y}, trans_{goal\ x}) \quad (6.14)$$

Die Gleichungen 6.13 und 6.14 zeigen die Berechnung des vollen Kreisbogenwinkels  $\alpha$ . Die Transformation der Zielposition mit der inversen Pose des Mittelpunktes, wandelt die 3D-, in eine 2D-Problematik um. Die Funktion *atan2* berechnet den Winkel im Bereich zwischen  $[-\pi : \pi]$ . Der normale Arkustangens nur zwischen  $[-\pi/2 : \pi/2]$ . Sobald  $\alpha < 0$  ist, muss nochmal  $2\pi$  auf ihn addiert werden, um den Kreisbogen nicht falsch herum auszuführen.

Wie bereits erwähnt, wird die Orientierung der Zwischenpose für den *SLERP*-Algorithmus benötigt. Hierzu wird der Winkel zu dieser Pose  $\beta$  analog zu den Gleichungen 6.13 und 6.14 ermittelt. Zur Umsetzung der `sample`-Funktion, wird nun erst zur Orientierung der Zwischenpose (am Winkel  $\beta$ ) interpoliert und anschließend zur Zielorientierung (am Winkel  $\alpha$ ).

### 6.1.3 Weginformationen

Jeder Wegpunkt kann zusätzlich noch weitere Informationen tragen, wie die gewünschte Endeffektorgeschwindigkeit oder weitere Optionen, die der reale Industrieroboter bei der Bewegung ausführen soll. Dazu ist eine Struktur vorhanden, die bei der Berechnung erkennt, welche Informationen dem Wegpunkt angegeben wurden.

## 6.2 Roboteraufgaben

Die Bahnplanung alleine reicht oft nicht aus, um einen Prozess mit einem Roboter auszuführen. Im Beispiel Lackieren oder Schweißen, muss das Werkzeug zur richtigen Zeit, am richtigem Ort geschaltet werden. Da solche Roboteraufgaben spezifische Roboterprogrammierung verwenden, ist es sehr aufwendig, alle möglichen Operationen verschiedener Roboterprogrammiersprachen abzudecken. Aus diesem Grund werden diese spezifischen Programmschnipsel in Form von Text in einer Roboteraufgabe definiert.

Beispielsweise sieht in der Roboterprogrammiersprache RAPID von ABB, der Code zum Warten auf ein Triggersignal am Eingang *di4* wie folgt aus:

```
WaitDI di4, 1;
```

Um eine Roboteraufgabe zu definieren, wird der Klartext solch eines Befehls beim Erstellen angegeben.

Jeder Wegpunkt verfügt noch über zwei weitere optionale Daten, nämlich die Roboteraufgaben für den Start und das Ziel. Die Bahnplanung ignoriert diese, sie werden aber bei der Ausführung am Roboter vor dem Starten des Wegpunktes, oder nach dessen Ende ausgeführt.

## 6.3 Bewegungsplanung zwischen Bahnen

Durch den *PtP*-Befehl (*Point to Point*) der Roboterprogrammierung, bewegt sich der Roboter von einer Konfiguration zur nächsten. Das geschieht durch synchrone Interpolation der Achsstellungen vom Start zum Ziel. In dieser Bahnplanung wird das Anfahren auf eine bestimmte Konfiguration im Raum benötigt, um den Start einer errechneten Bahn zu erreichen. Nach Durchführung der Bahn kann dann wieder zur Ausgangskonfiguration oder zur nächsten Bahn gefahren werden.

Um diese Bewegungen auf Kollision zu überprüfen, müssen die zwischenliegenden Konfigurationen getestet werden. Da die Endeffektorbewegung bei einem *PtP*-Befehl des Roboters nicht linear und schwer vorhersehbar ist, wird die Schrittweite durch eine maximale Achsbewegung dargestellt.

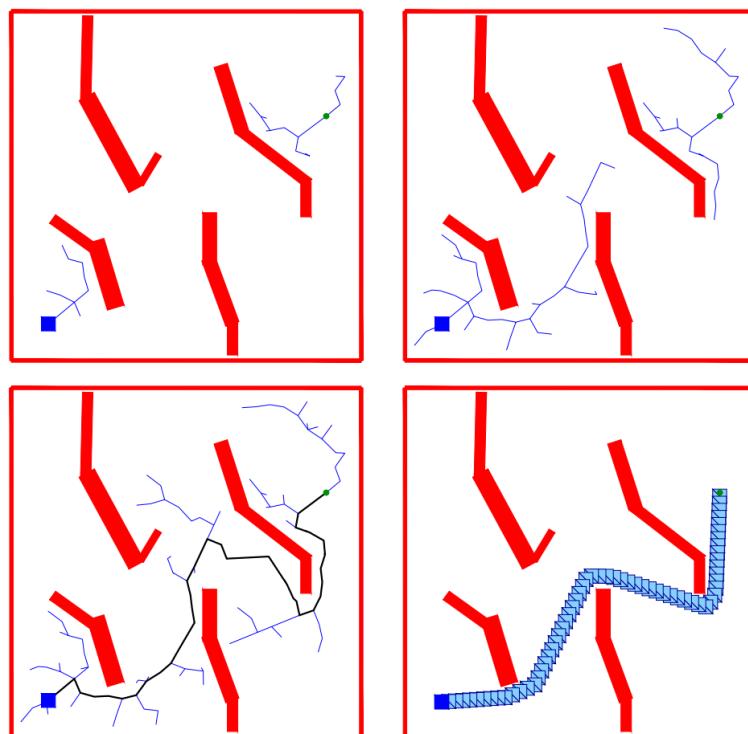
$$distance_{ptp} = \|\mathbf{config}_{goal} - \mathbf{config}_{start}\|_2 \quad (6.15)$$

Gleichung 6.15 zeigt die Berechnung der normierten Differenz der Konfigurationen. Bei einer gewählten Schrittweite von z. B.  $\Delta J = 0,1^\circ$  ist die Anzahl der Schritte  $n = \text{distance}_{\text{ptp}} / \Delta J$ . Durch lineare Interpolation werden anschließend Zwischenkonfigurationen errechnet, siehe Gleichung 6.16. Dabei ist  $s$  der aktuelle Schritt.

$$\text{config} = (\text{config}_{\text{goal}} - \text{config}_{\text{start}}) \cdot s/n + \text{config}_{\text{start}} \quad (6.16)$$

Falls dieser Weg eine Kollision hervorruft, wird eine andere Bewegung benötigt. In vielen Prozessen ist der Weg zu einer Bearbeitungsbahn hin, oder von ihr weg, nicht relevant, solange er nicht kollidiert. Im folgenden wird eine Technik vorgestellt, die diesen Weg kollisionsfrei ermitteln kann:

Für die Bewegungsplanung im Raum gibt es bereits mehrere Ansätze, wie den *RRT*-Algorithmus (*Rapidly-exploring Random Tree*). In diesem wächst eine Baumstruktur und breitet sich kollisionsfrei im Raum aus. Mit dem *RRT-Connect*-Algorithmus ist ein optimiertes Verfahren vorhanden, für den Weg von einem Start- zum Zielpunkt. Dabei wachsen zwei Bäume im Raum und versuchen sich zu verbinden [8, S. 2ff], siehe Abbildung 25.



**Abbildung 25:** *RRT-Connect*-Algorithmus bei einer 2D-Problemstellung [8, S. 4]

Eine Baumstruktur besteht aus einer strukturierten Sammlung von Knotenpunkten. Ein Knotenpunkt beinhaltet seine Position und eine Liste von Knotenpunkten, die aus ihm entstanden sind. Um Rechenzeit bei der Implementierung einzusparen, besitzt jeder Knotenpunkt noch einen Verweis auf seine Herkunft (bis auf den Startknoten).

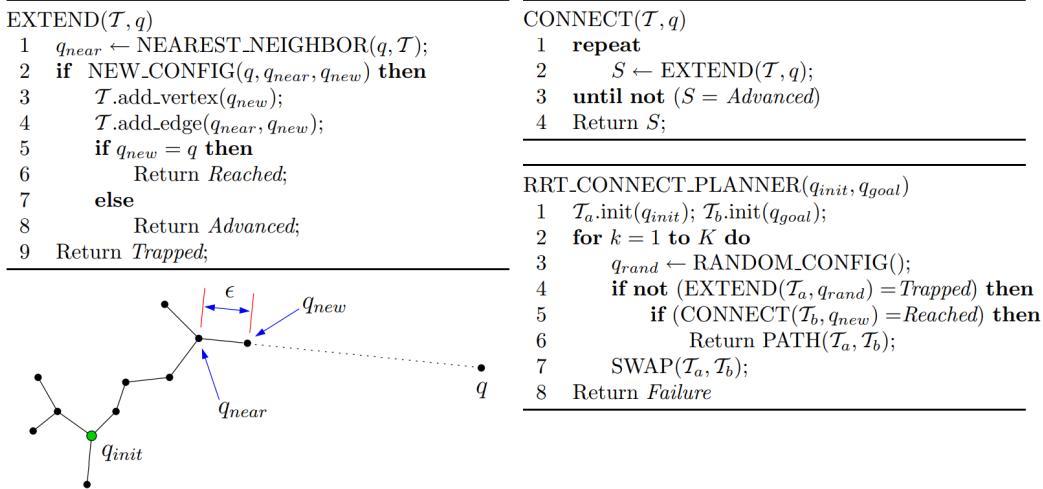


Abbildung 26: *RRT-Connect*-Pseudocode [8, S. 2f]

Abbildung 26 zeigt den Algorithmus von *RRT-Connect*. Die Funktion EXTEND nimmt als Parameter eine Baumstruktur  $\tau$  und eine Position im Raum  $q$ . Zunächst wird im Baum der Knoten gesucht, der der Position  $q$  am nächsten liegt ( $q_{near}$ ). Anschließend wird, falls keine Kollision eintritt, der Baum um den Knoten  $q_{new}$  ergänzt. Dieser zeigt in die Richtung von  $q$  mit einer Schrittweite von  $\epsilon$ . Die Funktion gibt dann zurück, ob  $q_{new}$  genau auf  $q$  liegt (*Reached*), ob  $q_{new}$  nicht erstellt werden konnte (*Trapped*), oder ob sie lediglich den Baum vergrößert hat (*Advanced*).

Der Hauptalgorithmus arbeitet dann beide Bäume abwechselnd ab: Nach der Initialisierung der Baumstrukturen, wird eine zufällige Position generiert  $q_{rand}$ . Wenn Baumstruktur  $\mathcal{T}_a$  durch die EXTEND Funktion ergänzt wurde, wird die Funktion CONNECT mit dem Baum  $\mathcal{T}_b$  aufgerufen. Diese ruft dann solange EXTEND auf, bis kein Knoten mehr ergänzt werden konnte, oder die Bäume zusammen gewachsen sind (Ziel der EXTEND Funktion ist  $q_{new}$  von Baum  $\mathcal{T}_a$ ). Wenn sie danach noch nicht zusammen gewachsen sind, wechselt sich Baum  $\mathcal{T}_a$  mit  $\mathcal{T}_b$  ab [8, S. 2ff].

Der Vorteil des *RRT-Connect*-Algorithmus ist, dass der Raum nicht bekannt sein muss. Es genügt zu wissen, ob eine Position kollidiert oder nicht. Die Schrittweite  $\epsilon$  sollte so gewählt werden, dass keine Kollision übersprungen wird, ähnlich zur bereits verwendeten Schrittweite  $\Delta J$ .

Der Algorithmus wird meist für 2D- und 3D-Problemstellungen verwendet. Er eignet sich jedoch auch für die Bewegungsplanung der sechs Achsen eines Industrieroboters.

Voraussetzung ist, dass in der Baumstruktur ein nahliegender Knoten gefunden werden kann und Kollisionen nicht durch die Schrittweite übersprungen werden. Durch die Verwendung einer Position mit sechs Dimensionen, arbeitet der *RRT-Connect*-Algorithmus in einem nicht darstellbaren, hyperraumähnlichem Gebilde.

Für die Implementierung wurde der Algorithmus angepasst. Vor der Ausführung wird zunächst überprüft, ob die einfache *PtP*-Bewegung des Roboters ausreicht und keine Kollision verursacht. Innerhalb des Algorithmus wird zwischen  $\epsilon$  und  $\Delta J$  unterschieden. Um Rechenzeit bei dem Aufbau der Baumstruktur einzusparen, liegt die Schrittweite  $\epsilon$  höher, als die Schrittweite zur Kollisionskontrolle. Bei der Anlegung eines neuen Knoten in der `EXTEND`-Funktion, wird mit der Schrittweite  $\Delta J$  der Weg zum neuem Knoten  $q_{new}$  überprüft, statt nur die Konfiguration aus  $q_{new}$ .

Wenn der Algorithmus Erfolg hat, wird der Weg in einer Liste von Konfigurationen gespeichert. Um die Liste zu vereinfachen, wird anschließend sequenziell versucht, einzelne Konfigurationen kollisionsfrei zu überspringen, ähnlich zu Abbildung 25, unten rechts.

Für die Roboterbibliothek wurde im Modul Bahnplanung eine Funktion erstellt, die genau diesen Algorithmus durchführt. Dabei nimmt die Funktion die Start- und Ziel-Konfiguration an und gibt die vereinfachte Liste der benötigten Konfigurationen zurück. Wenn kein Weg gefunden wurde, bleibt diese Liste leer.

Zur Ausführung am Roboter, werden dann die einzelnen Einträge der Liste, mit der *PtP*-Bewegung angefahren.

## 6.4 Trajektorie

In der entwickelten Roboterbibliothek sind einige Strukturen für die Bahnplanung vorhanden. Die Struktur einer Bahn besteht beispielsweise u. a. aus einer Liste von Wegpunkten. Dabei ist unerheblich, um welche Art von Wegpunkten es sich dabei handelt. Für die errechneten Wegpunkte ist eine weitere Struktur vorhanden, welche innerhalb der Bahnstruktur erstellt wird. In dieser Ergebnisstruktur werden die ermittelten Konfigurationen der Wegpunkte gespeichert, um sie später für die Ansteuerung des echten Industrieroboters nutzen zu können.

Angenommen es soll ein Bauteil mit mehreren Schweißnähten von dem Industrieroboter geschweißt werden, werden für die einzelnen Schweißnähte, einzelne Bahnen mit der Roboterbibliothek erstellt. Um die komplette Bahnplanung auszuführen, gibt es in der Bibliothek eine Funktion, die eine Trajektoriestruktur zurückgibt:

- **calculateTrajectory**

Diese Funktion nimmt eine Liste von Bahnen und optional die Start- und Zielkonfiguration an. Zuerst werden alle Bahnen der Liste berechnet. Bei Erfolg wird der Konfigurationsweg von der Ausgangskonfiguration, hin zur errechneten Startkonfiguration des ersten Wegpunktes der Bahn ermittelt, siehe Abschnitt 6.3. Anschließend wird nacheinander der Konfigurationsweg zwischen den Bahnen berechnet. Nach der letzten Bahn wird abschließend der Weg zur Zielkonfiguration bestimmt. Alle Ergebnisse werden dann in der Trajektoriestruktur zurückgegeben.

Die Trajektoriestruktur enthält bei erfolgreicher Berechnung alle Daten die notwendig sind, um den Roboter mit Befehlen zu programmieren. Dazu besteht sie aus einer Liste mit Konfigurationswegen und Bahnen. Die Bibliothek kann den Typ ermitteln und die entsprechenden Befehle erkennen.

## 6.5 Visualisierung

Um die berechneten Bahnen visuell darzustellen, enthält die entwickelte Roboterbibliothek Funktionen zur Anzeige der Bewegungen mit einstellbarer Geschwindigkeit. Darunter sind Funktionen zur Darstellung eines *PTP*-Befehls, eines Konfigurationsweges, eines Wegpunktes, einer Bahn und einer kompletten Trajektorie vorhanden.

Die Visualisierung wurde mithilfe von OpenRAVE umgesetzt, siehe Abbildung 27.



**Abbildung 27:** Visualisierung einer Trajektoriebewegung

## 7 Roboterschnittstelle

Um den Endeffektor eines Roboters im Raum bewegen zu können, muss dieser seine Achsen auf die entsprechenden Konfigurationen regeln. Dieser Regelkreis, zusammen mit der Leistungselektronik, befindet sich innerhalb der Robotersteuerung. Die benötigten Konfigurationen können, wie in Kapitel 4 beschrieben, mit der inversen Kinematik bestimmt werden.

Viele Roboterhersteller haben eigene Programmiersprachen für ihre Robotersteuerungen entwickelt, um das Verwenden ihrer Industrieroboter zu vereinfachen. Diese beinhalten u. a. Prozeduren zum Bewegen des Endeffektors im Raum, welche die inverse Kinematik bereits automatisch in der Steuerung berechnen. Solche Roboterprogrammiersprachen sind oft interpretierte Programmiersprachen, d. h. der Quellcode wird vorher nicht in Maschinencode kompiliert, sondern zur Laufzeit von der Steuerung gelesen und ausgeführt. Das hat u. a. den Vorteil, dass der Programmcode schnell gewechselt, zur Laufzeit angepasst und Funktionen anhand ihres Namens aufgerufen werden können.

Der für das Forschungsprojekt genutzte Industrieroboter ABB IRB 2600ID, wird mit der Programmiersprache ABB RAPID programmiert und kann nur über diese Befehle annehmen.

Um eine erfolgreich simulierte Bahnplanung auf dem Roboter auszuführen, bieten sich zwei Möglichkeiten an:

- **Generierung eines Roboterprogramms**

Die aus Kapitel 6 simulierte Trajektorie wird Schritt für Schritt in ein entsprechendes Roboterprogramm umgewandelt. Dieses wird anschließend auf den Roboter hochgeladen und ausgeführt.

- **Senden der Befehle zur Laufzeit**

Auf dem Roboter befindet sich ein vordefiniertes Programm, welches auf eine Netzwerkverbindung wartet. Wenn die Verbindung aufgebaut ist, werden die entsprechenden Befehle zum Roboter geschickt und von ihm ausgeführt.

### 7.1 Roboterprogrammiersprache ABB RAPID

Zur Programmierung von Industrierobotern, werden bei gleichbleibenden Prozessen normalerweise keine aufwendigen mathematischen Berechnungen, wie der linearen Algebra, benötigt. Dort reicht es aus, mit einfachen mathematischen Mitteln die Roboter zu programmieren. RAPID eignet sich bereits für solche Fälle, indem es einfache informatische Prinzipien beinhaltet, wie Variablen und einfache mathematische Funktionen. Komplexere Anwendungen benötigen eine externe Berechnung

und nutzen die Roboterprogrammiersprache meist lediglich zur Ausführung, so auch die in dieser Masterarbeit entwickelte Roboterbibliothek.

Für die Programmierung werden somit hauptsächlich die Bewegungsprozeduren verwendet. In RAPID sind zur Beschreibung der Konfiguration des Roboters zwei Darstellungen vorhanden:

- **jointtarget**

Hiermit wird die genaue Konfiguration des Roboters beschrieben:

$$[[0, -20, 60, 0, 60, 0], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]]$$

Das `jointtarget` besteht aus einer Liste der Achsstellungen in Grad, gefolgt von einer Liste externer Achsen. Wenn keine externe Achse verwendet wird, soll der Wert `9E9` betragen [15, S. 44].

- **robtarget**

Hiermit wird die Konfiguration indirekt über eine Pose bestimmt.

$$[[750, 300, 705], [0, 0, 1, 0], [0, -1, 0, 0], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]]$$

Das `robtarget` besteht zunächst aus einer Liste der  $x$ ,  $y$  und  $z$  Koordinaten der Pose in  $mm$ , gefolgt von einer Liste, die dessen Orientierung mit einem Quaternion beschreibt (mehr zu Posen, siehe Abschnitt 2.2). Die dritte Liste wird zur Vermeidung von Mehrdeutigkeiten verwendet. Der erste Parameter bestimmt, in welchem Quadranten sich Achse 0 befindet. Der zweite bestimmt Achse 3 und der dritte Achse 5. Der vierte Parameter wird für einen Roboter mit sieben oder mehr Achsen benötigt und wird für den in diesem Forschungsprojekt verwendeten Roboter mit sechs Achsen vernachlässigt. Die letzte Liste beschreibt ebenfalls die Stellung externer Achsen.

Werden Posen, bzw. `robtarget`, zur Beschreibung der Konfigurationen verwendet, kann die Konfigurationsüberwachung mit dem Befehl `ConfL\Off;` ausgeschaltet werden. In diesem Zustand wählt die Steuerung die passende Konfiguration zum Erreichen der Pose selbst, abhängig von der aktuellen Konfiguration [16, S. 61]. Das führt allerdings zu dem in Abschnitt 4.1 Abbildung 17 beschriebenen Problem. Mithilfe der oben erwähnten Konfigurationsbeschreibungen, können nun RAPID's Bewegungsprozeduren aufgerufen werden, die wichtigsten werden im folgenden kurz vorgestellt:

- **MoveJ**

Bewegt den Roboter in einer *PtP*-Bewegung zur angegebenen Pose, beschrieben als `robttarget`. Der Vorteil liegt in der Vermeidung von Singularitäten und einer schnellen Bewegung. Der Nachteil in dem nichtlinearem Weg.

- **MoveAbsJ**

Analog zu `MoveJ`, nutzt diese Prozedur zur Konfigurationsbeschreibung direkt die Konfiguration in Form des `jointtarget`. Da bereits durch die entwickelte Roboterbibliothek die genauen Konfigurationen bekannt sind, wird diese, statt der Prozedur `MoveJ`, verwendet.

- **MoveL**

Bewegt den Roboter in einer linearen Bewegung zur angegebenen Pose, beschrieben als `robttarget`. Werden Singularitäten angefahren, stoppt die Robotersteuerung.

- **MoveC**

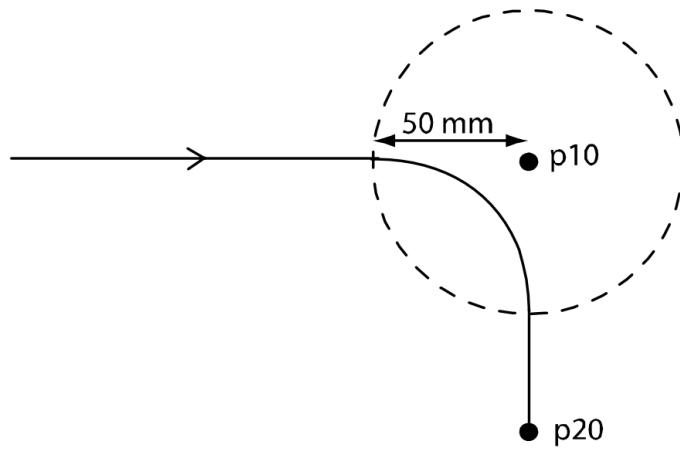
Bewegt den Roboter in einer kreisförmigen Bewegung zur angegeben Zielpose, über eine Pose die auf dem Kreisbogen liegt. Beide Posen werden als `robttarget` angegeben. Die Bewegung stoppt ebenfalls, sobald eine Singularität angefahren wird.

Zusätzlich zur Konfigurationsbeschreibung, benötigen die Bewegungsprozeduren noch die Angabe der Geschwindigkeit, der Zone und des verwendeten Werkzeugs (TCP). Optional kann auch ein anderes Koordinatensystem, statt des Roboterkoordinatensystems, angegeben werden.

Die Geschwindigkeit `speeddata` besteht aus vier Parametern, der Geschwindigkeit des Endeffektors im Raum in  $mm/s$ , seiner Drehgeschwindigkeit in  $^{\circ}/s$  und die entsprechenden Geschwindigkeiten externer Achsen [16, S. 1185ff].

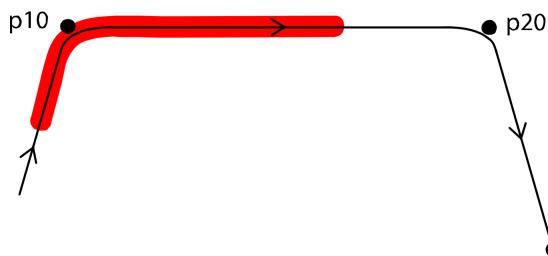
Ohne die Angabe einer Zone `zonedata`, fährt der Roboter die Zielkonfiguration genau an und verliert seine Geschwindigkeit. Für eine flüssigere Bewegung, kann eine Zone mit verschiedenen Parametern eingestellt werden. Abbildung 28 zeigt den Verlauf des Endeffektors bei einer eingestellten Zone von  $50mm$  [16, S. 1232-1237].

```
MoveL p10, v1000, z50, tool0;
MoveL p20, v1000, fine, tool0;
```



**Abbildung 28:** Zone zwischen zwei linearen RAPID-Bewegungen [15, S. 24]

Um Zonen zwischen den Bewegungen zu ermöglichen, besitzt der Interpreter von RAPID zwei Programmzeiger. Einer der beiden ist für die Ausführung der aktuellen Bewegung zuständig. Der andere liest bereits ein paar Schritte im Voraus, um so die Zonenbewegungen zu berechnen. Da dieser für die Ausführung von Ein- und Ausgabebefehlen zuständig ist, muss auf Synchronisierung geachtet werden, siehe Abbildung 29. Diese kann durch Deaktivieren der Zone, einer Wartezeit oder durch spezielle Bewegungsprozeduren erreicht werden.



**Abbildung 29:** Zu frühe Ausführung externer Werkzeuge [15, S. 29]

## 7.2 Generierung von RAPID-Programmen

Innerhalb der entwickelten Roboterbibliothek, wurde das Modul Postprozessor erstellt, welches sich um die Generierung von RAPID-Programmcode kümmert. Es besteht aus mehreren Funktionen die Text der passenden RAPID-Prozeduren aus z. B. Wegpunkten erstellt. Dieser Text kann im Anschluss in eine *mod*-Datei geschrieben und auf dem ABB IRB 2600ID ausgeführt werden.

Die beiden Funktionen `jointtargetFromConfiguration` und `robtargetFromPose` erzeugen aus einer Konfiguration, bzw. einer Pose mit ihrer errechneten Konfiguration, den passenden Text für die RAPID-Konfigurationsbeschreibung, siehe Abschnitt 7.1. Bei der Erstellung der `robtarget`, muss auf die richtige Berechnung der Konfigurationslage der Achsen 0, 3 und 5 geachtet werden, siehe Gleichung 7.1. Hierzu wird die Funktion `floor` verwendet, um eine reelle Zahl in die nächst kleinere ganze Zahl zu verwandeln.

$$c_i = \text{floor} \left( \frac{2 \cdot \text{config}_i}{\pi} \right) \quad (7.1)$$

Die Funktion `jointMovement` erzeugt den Text der RAPID-Prozedur `MoveAbsJ` für die übergebene Konfiguration. Der notwendige `jointtarget` wird mit der oberen Funktion zuvor erstellt.

Einzelne Wegpunkte lassen sich mit der Funktion `wayPointCommandBased` in Text der passenden RAPID-Prozeduren verwandeln. Bei einem linearem Wegpunkt wird die `MoveL`-Prozedur und bei einem zirkularem Wegpunkt die `MoveC`-Prozedur erzeugt. Bei letzterem wird aus dem Wegpunkt zusätzlich die auf dem Kreisbogen liegende Pose verwendet. Die notwendigen `robtarget` werden ebenfalls mit der oberen Funktion erstellt. Wenn ein Wegpunkt zum Start oder Ziel eine Roboteraufgabe enthält, wird ihr Text vor, bzw. hinter, den kommenden Prozeduren gesetzt. Somit enthält das RAPID-Programm Befehle zum Schalten von Werkzeugen, etc.

Für Wegpunkte ist die weitere Funktion `wayPointJointBased` vorhanden. Diese nutzt nicht direkt die Prozeduren `MoveL` und `MoveC`, sondern mehrmals die Prozedur `MoveAbsJ`. Es werden hierzu in einstellbarer Schrittweite von z. B.  $1\text{mm}$  Konfigurationen auf der Bahn des Wegpunktes erstellt, um somit den passenden Verlauf zu bestimmen. Für den Fall, dass später weitere Arten von Wegpunkten existieren, wie Spline-Wegpunkte, die nicht direkt in RAPID aufrufbar sind, wird innerhalb der Funktion `wayPointCommandBased`, diese Funktion verwendet.

Die Funktionen `pathCommandBased` und `pathJointBased` erstellen den Text für alle notwendigen RAPID-Prozeduren eines Pfades der Bahnplanung. Dazu nutzen sie die Funktionen `wayPointCommandBased`, bzw. `wayPointJointBased`, für die einzelnen Wegpunkte der Pfade.

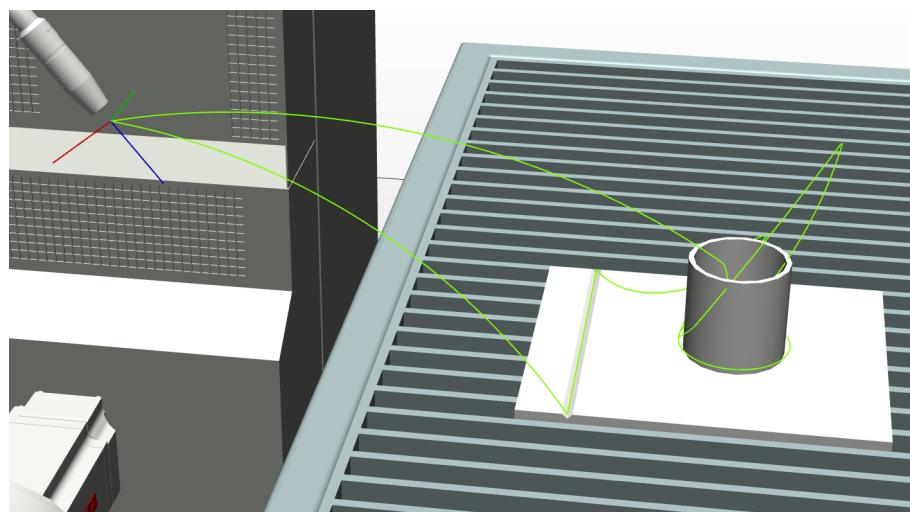
Eine berechnete Trajektorie kann analog mit dem Funktionen `trajectoryCommandBased` und `trajectoryJointBased` erstellt werden. Für die einzelnen Punkte der Konfigurationswege, wird die Funktion `jointMovement` verwendet.

Schlussendlich kann aus diesen Funktionen der gesamte RAPID-Programmcode erstellt werden. Neben den einzelnen Anweisungen gehören noch weitere Strukturelemente, wie der Modulname oder der Start der *main*-Prozedur. Die Funktionen `commandBasedProgram` und `jointBasedProgram` erzeugen hierzu den kompletten RAPID-Programmcode.

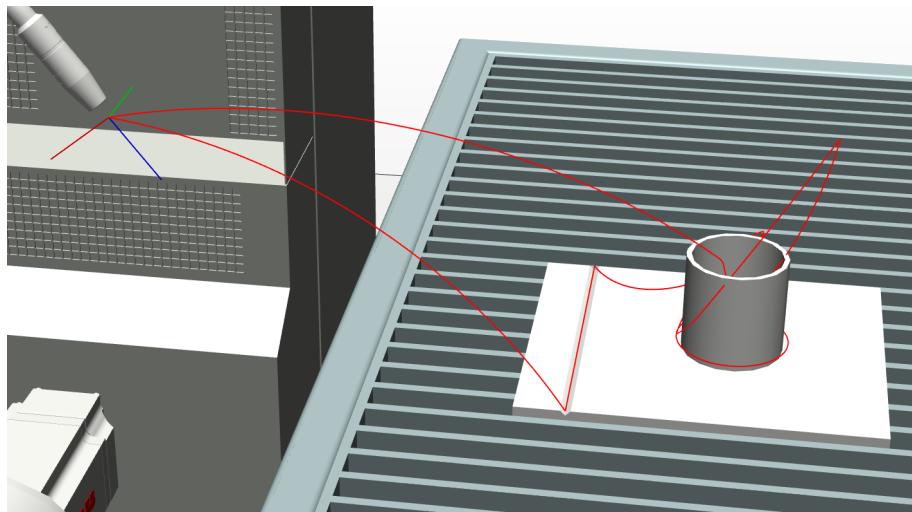
Um nicht von ABB abhängig zu werden, wurden alle Funktionen als Vorlage definiert, um somit auch andere Postprozessoren, anderer Roboter, erstellen zu können.

Die folgenden Abbildungen 30 - 32 zeigen den Verlauf des Endeffektors in der ABB RobotStudio Simulation. Das Schweißbauteil besteht aus einer linearen V-Naht, einer runden Kehlnaht um den Hohlzylinder und einer Punktschweißung in diesem. Für die V-Naht wurde ein linearer Wegpunkt erzeugt und für die Kehlnaht zwei zirkulare, um so einen ganzen Kreis zu beschreiben. Die Wege zwischen den Bahnen wurden durch die Bahnplanung automatisch kollisionsfrei ermittelt, siehe Abschnitt 6.3. Die berechnete Trajektorie wurde anschließend jeweils mit den Funktionen `commandBasedProgram` und `jointBasedProgram` in gültigen RAPID-Programmcode umgewandelt (siehe Quellcode 15 für das C++ Programm und Quellcode 16 für das generierte RAPID-Roboterprogramm in Anhang C).

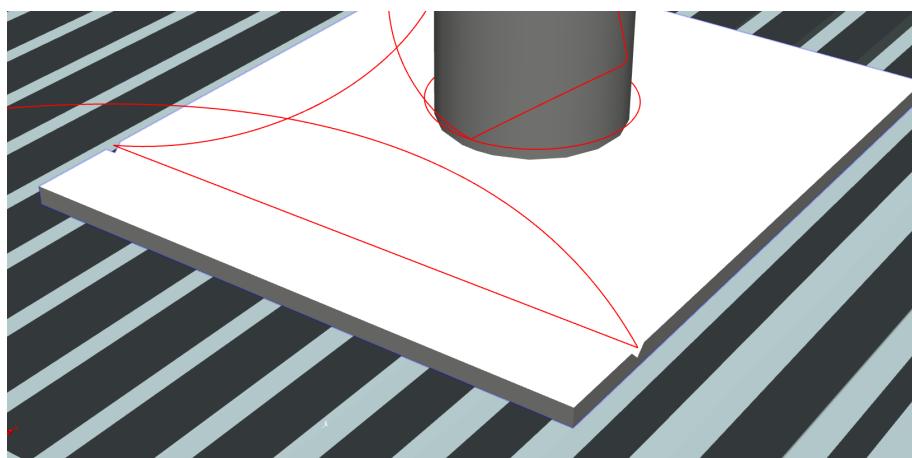
Bereits bei einer Schrittweite von 1 mm in der `jointBasedProgram` Funktion, ist ein Unterschied zur direkten Nutzung der Prozeduren `MoveL` und `MoveC` mit dem Auge nicht erkennbar.



**Abbildung 30:** Endeffektorbewegung bei einem generiertem RAPID-Programmcode mit Wegpunkten aus `MoveL` und `MoveC`



**Abbildung 31:** Endeffektorbewegung bei einem generierten RAPID-Programmcode mit Wegpunkten aus MoveAbsJ und einer Schrittgröße von 1 mm



**Abbildung 32:** Scheinbar flüssige Bewegung des Roboterendeffektors durch MoveAbsJ mit einer Schrittgröße von 1 mm

### 7.3 Kommunikation zur Laufzeit

Eine weitere Möglichkeit den Roboter zu steuern, liegt in dem direkten Senden von Befehlen. Für die Verbindung mit dem Computer können verschiedene Technologien verwendet werden. Die einfachste Möglichkeit bietet die Netzwerkprogrammierung, da so keine zusätzliche Hardware am Computer benötigt wird (mehr zur Netzwerkprogrammierung, siehe Abschnitt 2.5).

Für die Verbindung wird das TCP-Protokoll verwendet, um Datenverlust zu vermeiden und damit der Roboter die Befehle in richtiger Reihenfolge ausführt. Die TCP-Verbindung stellt *Byte-Streams* zum Senden und Empfangen am Computer und der Robotersteuerung zur Verfügung. Um über diese einzelne Nachrichten zu senden, wird ein eigenes Protokoll innerhalb der Anwendung verwendet.

### 7.3.1 Nachrichtenprotokoll

Für die entwickelte Netzwerkschnittstelle wurde folgendes Protokoll, in Anlehnung des Protokolls *vorbestimmte dynamische Nachrichtenlänge* (siehe Abschnitt 2.5.3), definiert:

**Tabelle 6:** Dynamisches Protokoll zur Ausführung von Roboterbefehlen

Byte Nr.	Verwendung
0	Länge des Befehlsnamens: $pn_{len}$
1 & 2	Länge der Nachrichtdaten: $d_{len}$
3 bis $(3 + pn_{len})$	Name des Befehls (1-80 Zeichen)
$(4 + pn_{len})$ bis $(4 + pn_{len} + d_{len})$	Daten der Nachricht (0-1024 Bytes)

Dieses Nachrichtenprotokoll liest somit zuerst die Größe des kommenden Prozedurnamens und der Datenlänge der Nachricht. Anschließend werden der Name und die Daten aus dem *Stream* gelesen. In ABB RAPID dürfen `strings` maximal 80 Zeichen lang sein und `rawbytes` maximal 1024 Bytes groß [16, S. 1195 und S. 1165].

Innerhalb von ABB RAPID lässt sich mit der Prozedur `CallByVar` anhand eines `strings` und einer Nummer eine andere Prozedur aufrufen. Beispielsweise würde der Befehl: `CallByVar "Proc", 2;` die Prozedur `Proc2` ausführen [16, S. 33]. Das ist möglich, da es sich bei ABB RAPID um eine interpretierte Programmiersprache handelt, welche die Namen der Elemente zur Laufzeit zur Verfügung hat.

Mithilfe dieser Prozedur kann nun eine Prozedur (im folgenden Befehl genannt), aus dem gelesenen Prozedurnamen  $pn$  einer Nachricht, ausgeführt werden. Da `CallByVar` als weiteren Parameter die Prozedurnummer benötigt, müssen alle durch Nachrichten ausführbaren Prozeduren mit einer Ziffer enden. In dem entwickeltem Roboterprogramm enden diese Befehle mit einer 0, u. a. `MoveAbsJ0`, `MoveL0` und `MoveC0`.

Auf diese Weise kann die entwickelte Roboterbibliothek, Befehle wie Bewegungsprozeduren zur Laufzeit in der Robotersteuerung ausführen.

Die Prozedur `CallByVar` bietet keine Möglichkeit, den aufgerufenen Prozeduren Parameter zu übertragen. Damit die Befehle dennoch auf die Daten der Nachricht zugreifen können, werden sie vorher global in der Variable `messageData` gespeichert. Diese wird dann von den verschiedenen Befehlen gelesen und verarbeitet.

Tabelle 7 zeigt den gekürzten Inhalt des `MoveAbsJ0`-Befehls, ohne dessen nachfolgenden Daten. Das erste Byte beschreibt, wie bereits erwähnt, die Länge des Befehlsnamens, die beiden nächsten die Länge der Daten (hier zusammen 24 für die sechs Achswinkel mit jeweils vier Bytes). Ab dem dritten Byte folgen die einzelnen Buchstaben als ASCII-Wert des Befehlsnamens. Ab dem elften Byte würden dementsprechend die 24 Bytes der Daten folgen.

**Tabelle 7:** Nachricht des MoveAbsJ0-Befehls

Byte Nr.	0	1	2	3	4	5	6	7	8	9	10	11...
Wert	8	24	0	77	111	118	101	65	98	115	74	...
ASCII				'M'	'o'	'v'	'e'	'A'	'b'	's'	'J'	

### 7.3.2 Ausführbare Befehle

Durch das entwickelte Protokoll ist es möglich, verschiedene Befehle anhand ihres Namens auszuführen und diesen unabhängige Daten, in Form von Bytes, zur Verfügung zu stellen.

Die von den Bewegungsprozeduren benötigten Posen und Konfigurationen, werden mit den Daten einer Nachricht `messageData` übermittelt. Die aufgerufenen Befehle behandeln diese Daten jeweils auf ihre eigene Art. Tabelle 8 zeigt die Datenstruktur um eine Konfiguration zu übertragen. Innerhalb des entwickelten Robotorprogramms kann mit der Funktion `jointFromData` aus den Daten ein `jointtarget` erstellt werden. Sie wird beispielsweise von dem Befehl `MoveAbsJ0` genutzt. Für die Darstellung reeller Zahlen verwendet ABB RAPID den Fließkommandatyp `float`, mit einer Größe von vier Bytes.

**Tabelle 8:** Datenstruktur zur Übertragung einer Konfiguration

Byte Nr.	Achstellung in °
0 – 3	Achse 0
4 – 7	Achse 1
8 – 11	Achse 2
12 – 15	Achse 3
16 – 19	Achse 4
20 – 23	Achse 5

Die Befehle `MoveL0` und `MoveC` benötigen eine, bzw. zwei Posen. Diese werden aus den Daten mit der Funktion `robFromData` erzeugt, siehe Tabelle 9. Damit innerhalb der Daten einer Nachricht mehrere Posen übertragen werden können, besitzen beide Funktionen einen optionalen Parameter `offset`, welcher die genutzten Bytes der Daten um dessen Wert verschiebt.

**Tabelle 9:** Datenstruktur zur Übertragung einer Pose

Byte Nr.	Verwendung
0 – 3	Position $X$ in mm
4 – 7	Position $Y$ in mm
8 – 11	Position $Z$ in mm
12 – 15	Orientierung $Q_w$
16 – 19	Orientierung $Q_x$
20 – 23	Orientierung $Q_y$
24 – 27	Orientierung $Q_z$
28 – 31	Konfigurationsquadrant Achse 0
32 – 35	Konfigurationsquadrant Achse 3
36 – 39	Konfigurationsquadrant Achse 5

Um die Endeffektorgeschwindigkeit und dessen Zonenoption für die Bewegungsbefehle einstellen zu können, existieren die Befehle `SetSpeed0` und `SetZone0`, welche aus den Daten die neuen Einstellungen speichern.

Um Werkzeuge und ähnliches zu steuern, kann das Roboterprogramm auf einfache Weise um Befehle, wie `TurnOnWeldingGun0`, erweitert werden. Diese können ebenfalls selbst entscheiden, wie sie mit den Daten umgehen.

### 7.3.3 Echtzeitproblematik

Die bisher vorgestellten Befehle würden theoretisch genügen, um den Roboter steuern zu können. Da die Netzwerkprogrammierung in der normalen Form nicht echtzeitfähig ist, können Probleme bei der Bewegung auftreten. Angenommen der Roboter befindet sich in einem Schweißvorgang, ist es kritisch, wenn die Steuerung zu lange auf den nächsten Befehl warten muss. Weitere Probleme können auftreten, wenn die Verbindung währenddessen unterbrochen wird und dieser Fall erst spät erkannt wird.

Um diese Probleme zu umgehen, kann entweder eine echtzeitfähige und sichere Übertragungsart verwendet werden, oder der Roboter erhält alle für den Schweißprozess nötigen Befehle im Voraus. Im folgenden wird letztere Möglichkeit vorgestellt:

Wenn der Ausführungszeitpunkt des eigentlichen Prozesses variiert kann, wird kein echtzeitfähiges System benötigt, solange die Robotersteuerung bereits alle notwendigen Befehle für den folgenden Prozess kennt. Um die Prozessbefehle ohne Ausführung zu übertragen, besitzt das entwickelte Roboterprogramm den Befehl `AddToList0`. Tabelle 10 zeigt die Datenstruktur dieses Befehls. Innerhalb dieser werden der später auszuführende Befehlsname mit dessen Daten übertragen.

**Tabelle 10:** Datenstruktur des AddToList0 Befehls

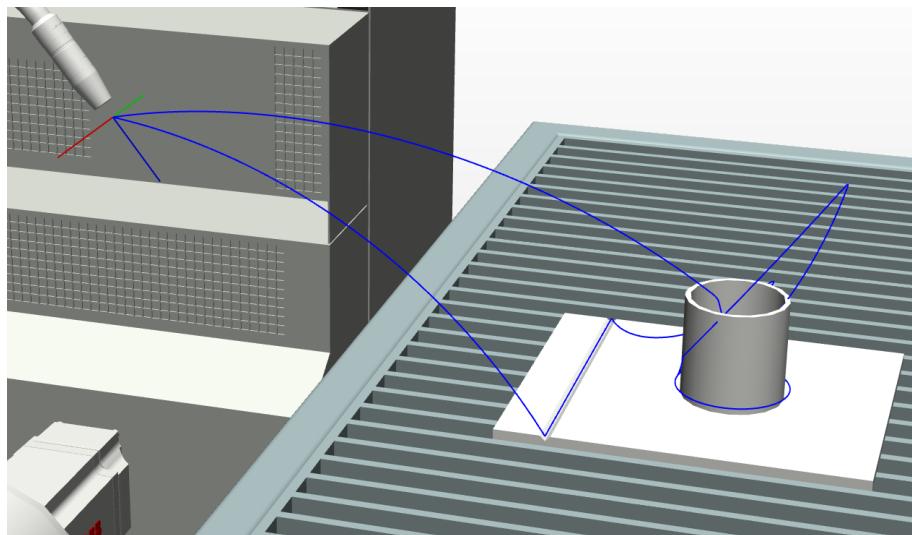
Byte Nr.	Verwendung
0	Länge des Befehlsnamen: $pn_{len}$
1 bis $pn_{len}$	Name des Befehls
(2 + $pn_{len}$ ) bis end	Daten des Befehls

In dem entwickeltem Roboterprogramm ist die Struktur `ListItem` vorhanden. Diese besteht aus dem `string` des Befehlsnamens und den `rawbytes` der Befehlsdaten. Von dieser Struktur existiert eine Liste, in die der Befehl `AddToList0` solch ein Befehlselement ans Ende anhängt.

Werden nun nach und nach Befehle über den `AddToList0` Befehl gesendet, bildet diese Liste die Befehle in der richtigen Reihenfolge ab.

Der Befehl `StartList0` arbeitet nach Aufruf nun genau diese Liste ab, indem die Daten aus der aktuellen Struktur in die `messageData` kopiert werden und der Befehl mit der Prozedur `CallByVar` aufgerufen wird. Sobald die gesamte Liste abgearbeitet wurde, wird der Listenzähler wieder zurückgesetzt und es kann eine neue Liste erstellt werden.

Abbildung 33 zeigt den Verlauf des Endeffektors bei der Steuerung des Roboters zur Laufzeit (siehe Quellcode 15 für das C++ Programm und Quellcode 17 für das entwickelte RAPID-Roboterprogramm der Netzwerkschnittstelle in Anhang C).

**Abbildung 33:** Endeffektorbewegung bei der Kommunikation zur Laufzeit

## 8 Eingabeschnittstelle

Die bisher vorgestellten Module der entwickelten Roboterbibliothek genügen bereits, um einen Roboter kollisionsfrei zu programmieren. Die Trennung der Prozessbahnen zusammen mit der Berechnung der Erreichbarkeit, bringen dieser Roboterbibliothek im Vergleich zur normalen Roboterprogrammierung Vorteile.

Dennoch ist die Programmierung aufwendig. Alleine die Initialisierung der Module nimmt viel Platz ein, obwohl sich die Roboterdaten und die Umgebung nur selten im Betrieb ändern. Wenn an dem selben Industrieroboter weitere Programme erstellt werden sollen, bietet es sich an, den gesamten Initialisierungsprozess innerhalb einer Funktion zu verarbeiten.

Ein größerer Nachteil ist die Erstellung von Posen und Wegpunkten. Bisher müssen für alle Wegpunkte, Start und Ziel angegeben werden. Dabei ist der Start des nächsten Wegpunktes einer Bahn zwingend das Ziel der letzten. Außerdem gestaltet sich die Bearbeitung von Posen aufwendig. Für eine Eingabeschnittstelle bietet es sich an, einen Wegpunkt in einer Zeile, zusammen mit der neuen Pose des Ziels, erstellen zu können.

(Anhang C zeigt im Quellcode 15 das Programmierbeispiel ohne Eingabeschnittstelle aus Kapitel 7)

In anderen Roboterprogrammiersprachen, oder der Programmierung von NC-Maschinen mit G-code, ist dies ebenfalls möglich. Dort wird das Ziel des Wegpunktes, zusammen mit der Art, in einer Zeile angegeben. In ABB RAPID erzeugt die Prozedur `MoveL p1, v1000, z30, tool0;` eine lineare Bewegung vom Ziel des letzten Wegpunktes, hin zur Pose `p1` [16, S. 264].

Ziel der Entwicklung der Eingabeschnittstelle ist es, den Programmieraufwand zu verringern. Deswegen wird die Initialisierung mithilfe der Funktion `setup` verarbeitet und zur Beschreibung von Bahnen, die Definition von Wegpunkten in einer Zeile ermöglicht.

### 8.1 Gestaltung einer prozeduellen Eingabeschnittstelle

Um die Eingabe eines Wegpunktes mit einer Pose in einer Zeile zu ermöglichen, wird eine intern gespeicherte Pose benötigt. Diese wird bei einer Wegpunkterstellung auf dessen Ziel gesetzt, sodass die Eingaben nur noch die Ziele der Wegpunkte benötigen und der Start aus der gespeicherten Pose entnommen werden kann.

In der Programmiersprache C++ gibt es verschiedene Varianten solch einer Eingabeschnittstelle zu implementieren. Um die eigentliche Bahndefinition so gering wie möglich zu gestalten, werden globale Prozeduren verwendet. Die Eingabeschnitt-

stelle verwendet den Namensraum `ii` (*input interface*), damit dessen Namen nicht mit anderen globalen Namen in Konflikt geraten. Generell ist die gesamte entwickelte Roboterbibliothek in dem Namensraum `pf` (*path finder*) und somit die Eingabeschnittstelle in `pf::ii`.

Wird die Roboterbibliothek verwendet, können in den Quelldateien die diese verwenden, die Namensräume aufgelöst werden. So muss nicht vor jeder Klasse oder Funktion der Namensraum erwähnt werden.

Wenn sich von einem Bahnschritt zum nächsten z. B. nur der *X*-Wert ändert, soll die neue Eingabeschnittstelle es ermöglichen, auch nur *X* für die Bewegungsprozeduren angeben zu müssen. Dieses Verhalten besitzt auch G-code, indem dort hinter dem Bewegungsbefehl explizit angegeben werden muss, um welchen Parameter es sich handelt, z. B.: `G1 X20`.

### 8.1.1 Optionale Eingabeparameter

Eine Bewegungsprozedur, wie die der G-code-Programmierung, ist in C++ nicht möglich. Innerhalb von C++ benötigt die Prozedur Parameter in einer vorbestimmten Reihenfolge. Ohne großen Aufwand, lässt sich die Parameterreihenfolge *x, y, z, orientation* verwenden. Somit sähe ein Bewegungsauftrag für eine lineare Bewegung wie folgt aus: `moveL(10,20,30,orientation);`.

Wie zu erkennen ist, besitzen die einzelnen Positionswerte keine Angabe, um welche Dimension es sich handelt. Somit ist es mit dieser Art nicht möglich, die optionalen Eingabeparameter zu realisieren, um z. B. nur *Z* zu verändern.

Eine Alternative besteht darin, die genutzten Eingabeparameter in den Prozedurnamen zu setzen, z. B.: `moveLyZ(20,30);`.

Der Nachteil dieser Art liegt in dem Erstellen der vielen verschiedenen Prozeduren und der so eingehenden sehr schlechten Wartbarkeit.

Text als Eingabeparameter bietet eine weitere Alternative. In dem übergebenen Text kann, wie in G-code, die Art mit angegeben werden, z. B.: `moveL("X20");`.

Der Nachteil liegt in dem hohen Programmieraufwand zur Realisierung und dem aufwendigeren Anwenden von Variablen. Ist beispielsweise die Variable `weldZ=30` vorhanden, sähe der Aufruf wie folgt aus: `moveL("X10 Z" s + to_string(weldZ));`.

Zur Umsetzung der optionalen Eingabeparameter der Eingabeschnittstelle wurde die Programmiermethode Polymorphie verwendet.

Diese ermöglicht es, eine abstrakte Mutterklasse zu erstellen, hier `Value`. Nun können von dieser Klasse, Kinderklassen erstellt werden, die die Eigenschaften der Mutter erben. Die Klasse `ValuePosition` beinhaltet beispielsweise einen der Parameter *X*,

$Y$  oder  $Z$ , leitet von `Value` ab und kann deswegen auch wie `Value` behandelt werden. Jedes Kind kann nun die Methode `changePose` je nach Art überschreiben. Z. B. ändert `ValuePosition` einen der Parameter  $X$ ,  $Y$  oder  $Z$  der aktuellen Pose.

Die Bewegungsprozeduren besitzen nun alle die gleichen Parameter, nämlich mindestens ein `Value` und maximal vier. Somit kann auch nur ein einzelner Wert bei einer Wegpunktangabe geändert werden oder, zusammen mit der Orientierung, alle vier.

Innerhalb der Bewegungsprozeduren wird nun die gespeicherte Pose als Startpose verwendet. Anschließend wird die `changePose` Methode aufgerufen, welche anhand der Kinderimplementierung die gespeicherte Pose bearbeitet. Abschließend wird der Wegpunkt mit der geänderten gespeicherten Pose als Zielpose erstellt und in die aktuelle Bahn angehängt.

Um die Erstellung der Kinderklassen wie `ValuePosition` zu vereinfachen, wurden Funktionen wie `x` erstellt, die die passende erstellte Klasse zurückgeben. Somit sieht der Aufruf einer Bewegungsprozedur wie folgt aus: `moveL(x(10), z(30));`, wobei die Reihenfolge von `x` und `z` keine Rolle spielt.

Im Vergleich zur Textvariante, bietet diese Möglichkeit eine einfachere Verwendung von Variablen, z. B.: `moveL(x(10), z(weldZ));`.

Für die Orientierungsangabe wurden verschiedene Klassen und Funktionen erzeugt: `euler` für Eulersche Winkel, `rotaxis` für Rotationsachsen, `quat` für Quaternionen und `rotmat` für Rotationsmatrizen.

Mehr zu den verschiedenen Arten der Orientierungsdarstellung und den benötigten Parametern, siehe Abschnitt 2.2.

Damit auch relative Koordinaten und Bewegungen genutzt werden können, sind folgende Funktionen ebenfalls in der Eingabeschnittstelle vorhanden:

`addX`, `addY`, ..., `addEuler`, ...

Im Folgenden werden die wichtigsten Bewegungsprozeduren kurz erläutert:

- **start**

Versetzt die intern gespeicherte Pose, ohne eine Bahn zu bearbeiten. Somit lassen sich die Startposen der Bahnen einstellen.

- **moveL**

Erzeugt einen linearen Wegpunkt und hängt ihn an die aktuelle Bahn an. Für die Startpose wird die intern gespeicherte Pose verwendet, anschließend wird diese versetzt und als Zielpose genutzt.

- **moveCvia**

Um Konsistenz der Bewegungsprozeduren einzuhalten, wurden die zwei benötigten Posen eines zirkularen Wegpunktes, in zwei Prozeduren aufgeteilt. Diese speichert lediglich die Zwischenpose des Wegpunktes ab, ohne die Bahn zu bearbeiten.

- **moveCto**

Zusammen mit der Pose aus der zuvor aufgerufenen Prozedur `moveCvia`, wird nun ein zirkularer Wegpunkt erzeugt und in die aktuelle Bahn angehängt.

- **nextPath**

Diese Prozedur nimmt im Vergleich zu den bisher vorgestellten, keine Parameter an. Sie schließt die aktuelle Bahn ab und erzeugt eine neue.

Die Roboterprogrammiersprachen bieten die Möglichkeit das Koordinatensystem zu wechseln. So lassen sich z. B. alle Posen vom Arbeitstisch oder dem Werkstück selbst angeben.

In der entwickelten Eingabeschnittstelle ist die Prozedur `setCoordinationSystem` vorhanden, welche eine Pose als Parameter nimmt und intern speichert. Mit dieser Pose werden vor der Generierung von Wegpunkten alle Posen transformiert. Somit lassen sich die Bewegungsprozeduren auch mit einem anderem Koordinatensystem, als das des Roboters verwenden, z. B. das eines Schweißbauteils.

Zur weiteren Vereinfachung befinden sich ebenfalls die Prozeduren `useMilliMeters`, `useMeters`, `useDegrees` und `useRadians` in der Eingabeschnittstelle. Wie deren Namen schon andeuten, können sie benutzt werden, um alle folgenden Eingaben in Millimetern anzugeben, statt in Metern, bzw. in Grad, statt in Radian.

Durch die optionalen Eingabeparameter, der möglichen relativen Eingabe und den weiteren Vereinfachungen, bietet die entwickelte Eingabeschnittstelle bereits Vorteile zu anderen Roboterprogrammiersprachen.

### 8.1.2 Weginformationen

Die Roboterrendeffektorgeschwindigkeit und verwendete Zonenoption lassen sich über die Prozeduren `speed` und `zone` einstellen. Dabei nehmen diese die Geschwindigkeit und Zone entweder in  $m/s$  und  $m$  oder in  $mm/s$  und  $mm$  an, abhängig von der verwendeten Einheit, siehe oben.

Theoretisch hätten diese Wegpunktinformationen auch über die oben vorgestellte Value-Technik realisiert werden können. In diesem Fall sähe eine Bewegungsprozedur beispielsweise wie folgt aus:

```
moveL(y(20), z(30), euler(0,0,90), speed(100), zone(5));.
```

In der entwickelten Eingabeschnittstelle wurde darauf aber zur Übersichtlichkeit der Bahnbeschreibung verzichtet. Innerhalb der Bewegungsprozeduren beschreiben die Parameter somit nur die Zielpose.

Der Aufruf einer der beiden Prozeduren sorgt somit dafür, dass die folgenden Bewegungsprozeduren die neue Einstellung nutzen und daraus die Wegpunktinformationen für die Geschwindigkeit und Zone erstellen.

### 8.1.3 Roboteraufgaben

Wie die Weginformationen, werden sie mithilfe einer eigenen Prozedur eingestellt. Die Prozedur `task` nimmt als Eingabe eine Roboteraufgabe (mehr zu Roboteraufgaben, siehe Abschnitt 6.2). Diese wird entweder auf den Start des nächsten Wegpunktes gesetzt, oder an das Ende des letzten, wenn die Bahn nach dem Aufruf von `task` beendet wird.

Sobald die entsprechende Bahn, bzw. Bahnen definiert wurden, können sie mit der Funktion `getPaths` zurückgegeben werden. Mit dieser Liste von Bahnen wird anschließend die Trajektorie berechnet und der Roboter gesteuert (mehr zur Trajektorie, siehe Abschnitt 6.4). Die Prozedur `resetPaths` kann abschließend dazu genutzt werden, die intern gespeicherte Bahnenliste zu leeren, um so neue Eingaben vornehmen zu können.

Quellcode 18 aus Anhang D zeigt das neue Programm, mit dem Beispiel des Schweißbauteils aus Kapitel 7.

Eine komplette Dokumentation der Roboterbibliothek und dessen Module befindet sich in Anhang E.

## 8.2 Anbindung zu einer interpretierten Programmiersprache

Ziel dieser Masterarbeit ist es, eine Software zu entwickeln, die zuvor generierte Bahnen nutzt, um damit einen Industrieroboter über ein Bauteil kollisionsfrei zu manövrieren.

Die entwickelte Roboterbibliothek kann durch die Eingabeschnittstelle hingegen ebenfalls verwendet werden, um Roboterprogramme aus vordefinierten, manuell eingegebenen, Bahnen zu erstellen.

In diesem Fall zeigt die Programmiersprache C++ ihre Schwächen. Zum einen kann die Kompilierung, um das Programm zu erstellen, einige Zeit in Anspruch nehmen, selbst wenn nur ein Wert geändert wird. Zum anderen ist die Programmierung für

unerfahrene Programmierer aufwendiger.

Aus diesem Grund wurde für die entwickelte Eingabeschnittstelle eine Bibliothek-Anbindung für die Programmiersprache Python erstellt.

### 8.2.1 Die Programmiersprache Python

Python ist eine dynamisch typisierte, objektorientierte, interpretierte Programmiersprache [35]. Das heißt, dass Klassen definiert werden können, aber bei der Initialisierung nicht angegeben werden müssen. In C++ sieht beispielsweise die Initialisierung einer Ganzzahl wie folgt aus: `int a = 10;`.

Das Gleiche wird in Python durch folgende Codezeile erreicht: `a = 10.`

Da Python interpretiert ist, fällt die Kompilierung des Programmes quasi weg und das Programm wird sofort ausgeführt. Tatsächlich wird aus dem Python-Programm, Bytecode erzeugt. Die hierfür verbrauchte Rechenzeit ist aber kaum bemerkbar.

Mit Python lässt sich die entwickelte Eingabeschnittstelle leichter zur Erstellung eines Roboterprogrammes nutzen, da die Bahnen schneller getestet werden können und die Programmiersprache einfacher zu nutzen ist.

Ein weiterer Vorteil von Python liegt in dem Live-Interpreter. Mit ihm lassen sich Python-Befehle direkt eingeben und sofort ausführen. So können z. B. Posen leichter getestet werden, indem sofort dargestellt wird, wo sie sich in der virtuellen Umgebung befinden.

### 8.2.2 Python-Anbindung

Um die Eingabeschnittstelle zur Verfügung zu stellen, wurde die C++-Bibliothek *Boost.Python* verwendet [17]. Mit ihr lässt sich eine Python-Bibliothek erstellen, indem Funktionen und Prozeduren anhand ihres Namens verknüpft werden.

Während des Verfassens dieser Masterarbeit wird aus Zeitgründen nur die Eingabeschnittstelle in Python eingebunden. Für eine universellere Nutzung sollte die gesamte entwickelte Roboterbibliothek mit allen Klassen, wie Posen, etc., in Python verfügbar sein. So wären die Vorteile beider Programmiersprachen vorhanden. Das sind die Geschwindigkeit und statische Sicherheit aus C++, zusammen mit der Möglichkeit schnell und einfach, mithilfe von Python, Roboterprogramme zu erstellen. Der folgende Quellcode 1 zeigt das gekürzte Programm des Schweißbauteils mit der Python-Anbindung. Die ungetkürzte Version, die nicht nur die V-Naht abbildet, befindet sich im Anhang D unter Quellcode 19.

#### Quellcode 1: Gekürztes Python-Programm des Schweißbauteils

```
1 from pypf import *
2 setupIRB2600ID()
```

```

4  useMilliMeters()
5  useDegrees()

7  speed(50)
8  zone(5)
9  turnOn = createTask("TurnOnWeldingGun", "WaitTime \\InPos,0;\n! SetDO do1, 1;")
10 turnOff = createTask("TurnOffWeldingGun", "WaitTime \\InPos,0;\n! SetDO do1, 0;")

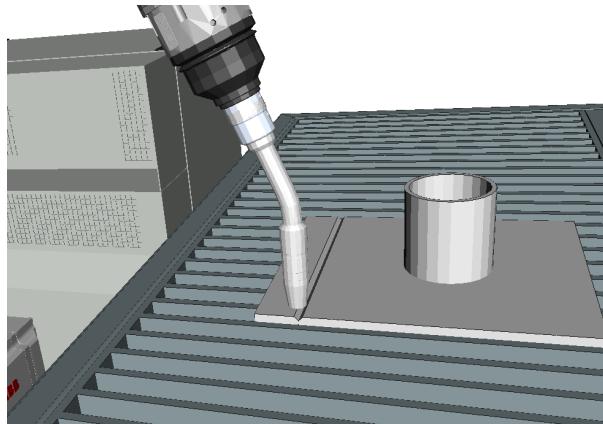
12 setCoordinationSystem(700, 0, 710, 0, 0, 1, 0)

14 start(x(-50))
15 task(turnOn)
16 moveL(y(300))
17 task(turnOff)

19 if calculate():
20     visualize()

```

Um das korrekte Einstellen einer Pose zu vereinfachen, wurde die Python-Anbindung um die Prozedur `show` ergänzt. Durch sie wird eine zufällige Konfiguration zum Erreichen der aktuellen Pose in der Visualisierung dargestellt, siehe Abbildung 34.



**Abbildung 34:** Visualisierung nach Aufruf der `show`-Prozedur

Wenn die Pose auch mit Kollision nicht erreicht werden kann, bleibt der Roboter in der Visualisierung stehen. Der folgende Quellcode 2 zeigt die Schritte von Abbildung 34, im Python-Live-Interpreter.

#### Quellcode 2: Live-Interpreter-Eingaben zur Darstellung einer Pose

```

1 $ python
2 Python 2.7.13 (default, Nov 24 2017, 17:33:09)
3 [GCC 6.3.0 20170516] on linux2
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>> from pypf import *
6 >>> setupIRB2600ID()
7 >>> useMilliMeters()
8 >>> useDegrees()
9 >>> setCoordinationSystem(700, 0, 710, 0, 0, 1, 0)
10 >>> start(x(-50))
11 >>> show()

```

## 9 Ergebnisse

Zur Auswertung der entwickelten Roboterbibliothek, wird sie anhand einiger Beispiele getestet. Während des Verfassens dieser Masterarbeit wurde bereits eine virtuelle Schweißumgebung erstellt. Da das Beispiel Schweißroboter die verschiedenen Module der Roboterbibliothek benötigt, eignet sich dieses ebenfalls für die Auswertung.

Jedes Beispiel enthält dabei jeweils den Programmabschnitt, der die Prozessbahnen beschreibt und die Rechendauer als Mittelwert mit Standardabweichung, ermittelt aus zehn Ausführungen. Nach erfolgreich berechneter Trajektorie, wird sie jeweils über den generierten RAPID-Roboterprogrammcode und der Laufzeitverbindung in ABB Robotstudio getestet (mehr zur Trajektorie, siehe Abschnitt 6.4, mehr zur Roboterschnittstelle, siehe Kapitel 7).

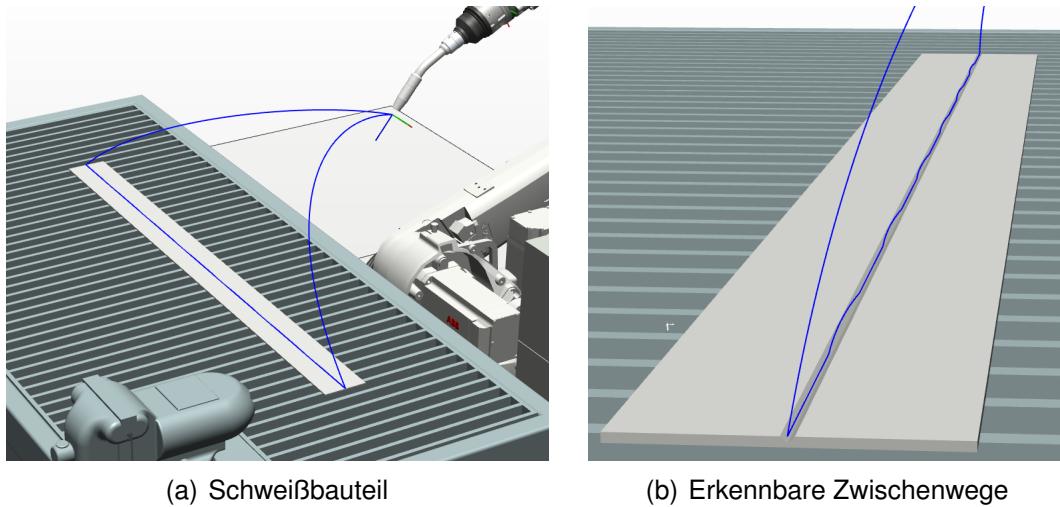
Zur Darstellung der Bewegung des Roboters, enthalten die Abbildungen den Verlauf des Roboterendeffektors. Die Berechnung fand auf einem Intel Core i5 Vier-Kern-Prozessor, mit 3,8 GHz statt.

### 9.1 Einfache Schweißbauteile

Die folgenden Schweißbauteile weisen übliche, einfache Geometrien und Schweißarten auf.

Das in Kapitel 7 verwendete Schweißbauteil, bestehend aus einer V-Naht, einer runden Kehlnaht und der Punktnaht innerhalb des Hohlzylinders, wurde bereits erfolgreich getestet. Die für die Berechnung notwendige Rechendauer beträgt im Durchschnitt  $\bar{x} = 21,387\text{ s}$  mit einer Standardabweichung von  $\sigma = 6,348\text{ s}$ .

Das folgende Beispiel zeigt eine lange V-Naht, bestehend aus sieben Schweißnähten. Mittelwert und Standardabweichung der Berechnungsdauer betragen:  $\bar{x} = 0,540\text{ s}$  und  $\sigma = 0,158\text{ s}$ .



**Abbildung 35:** Endeffektorbewegung einer langen V-Naht, bestehend aus mehreren Schweißnähten

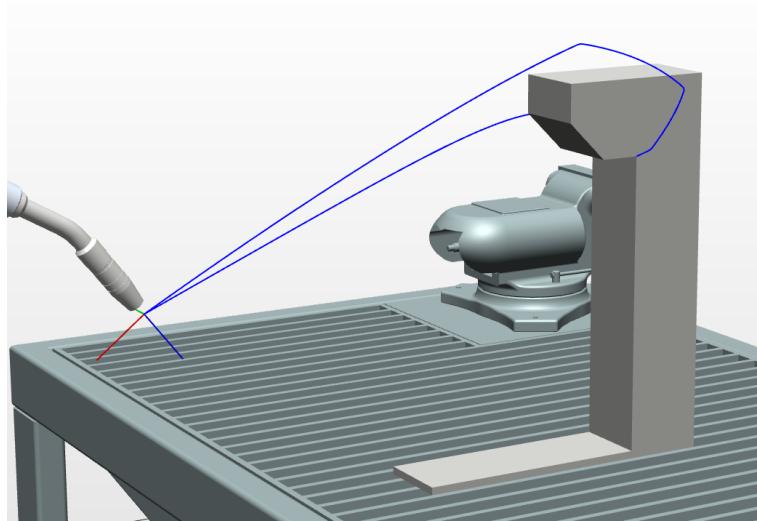
Wie in Abbildung 35b erkennbar ist, wird zwischen den Schweißnähten kein linearer Weg zur kollisionsfreien Erreichung der nächsten Prozessbahn ermittelt. Das liegt daran, dass der Roboter so weniger mit seinen Achsen verfahren muss. Der folgende Quellcodeabschnitt 3 zeigt die Beschreibung der Prozessbahnen in C++.

#### Quellcode 3: Codeausschnitt der langen V-Naht

```

1  useMilliMeters();
2  useDegree();
3  // Position, Orientation as Quaternion
4  //           x,      y,      z, w, i, j, k
5  setCoordinationSystem(Pose{900, -650, 703, 0, 0, 1, 0});
6  start(x(50));
7  for(var i=0; i<7; ++i) {
8      task(turnOn);
9      moveL(addY(100));
10     task(turnOff);
11     if(i<6) {
12         nextPath();
13         start(addY(100));
14     }
15 }
```

Das nächste Beispiel besteht aus einer Kehlnaht in horizontaler Überkopf-Position. Der Mittelwert und die Standardabweichung zur Berechnung betragen:  $\bar{x} = 13,802\text{ s}$  und  $\sigma = 4,117\text{ s}$ .

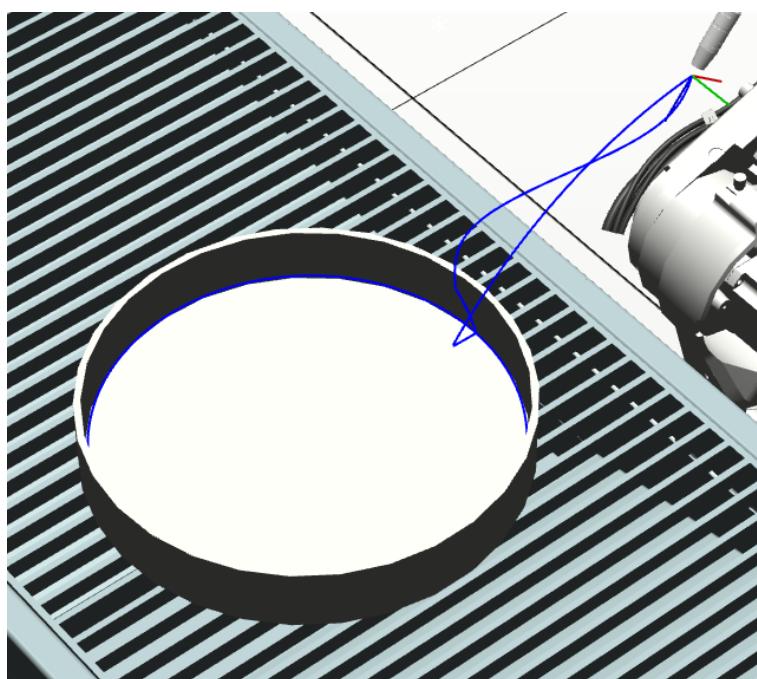


**Abbildung 36:** Endeffektorbewegung einer Kehlnaht in horizontaler Überkopf-Position

**Quellcode 4:** Codeausschnitt der Kehlnaht in horizontaler Überkopf-Position

```
1 useMilliMeters();
2 useDegree();
3 setCoordinationSystem(Pose{700, 0, 700, 0, 0, 1, 0});
4 start(x(-300), z(-400), euler(0,-135,180));
5 task(turnOn);
6 moveL(y(100));
7 task(turnOff);
```

Im folgenden Beispiel wird ein breiter Hohlzylinder mit einer langen Kehlnaht von innen geschweißt. Die Rechendauer beträgt im Durchschnitt  $\bar{x} = 4,351\text{ s}$  mit einer Standardabweichung von  $\sigma = 0,474\text{ s}$ .



**Abbildung 37:** Endeffektorbewegung einer inneren Kehlnaht mit einer Schweißnaht

Im dazugehörigen Quellcodeabschnitt 5 wurden zwei zirkulare Wegpunkte erzeugt. Das liegt daran, dass die benötigte Zwischenpunktpose weniger als  $\pi$ , bzw.  $180^\circ$  von der Start- und Zielpose entfernt sein muss (mehr zu zirkularen Wegpunkten, siehe Abschnitt 6.1.2). Bei einer zirkularen Bewegung von exakt  $2\pi$ , bzw.  $360^\circ$  werden dementsprechend mindestens zwei benötigt.

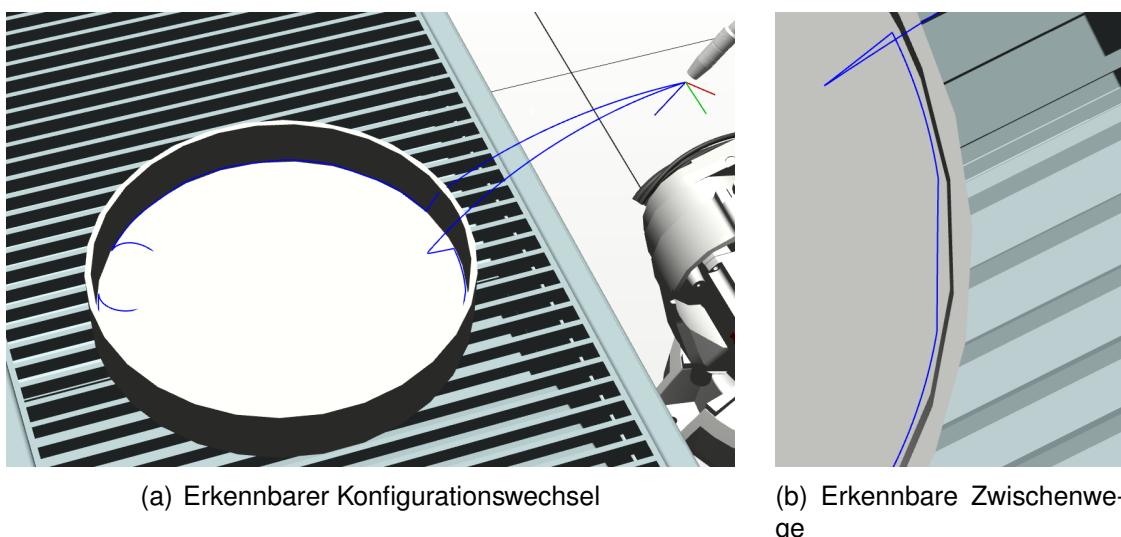
**Quellcode 5:** Codeausschnitt der inneren Kehlnaht mit einer Schweißnaht

```

1  useMilliMeters();
2  useDegree();
3  setCoordinationSystem(Pose{1000, 0, 710, 0, 0, 1, 0});
4  start(x(295), z(-8.66025), euler(0, 30, 0));
5  task(turnOn);
6  moveCvia(x(0), y(295), addEuler(0, 0, 90));
7  moveCto(x(-295), y(0), addEuler(0, 0, 90));
8  moveCvia(x(0), y(-295), addEuler(0, 0, 90));
9  moveCto(x(295), y(0), addEuler(0, 0, 90));
10 task(turnOff);

```

Das gleiche Beispiel mit mehreren Schweißnähten zeigt Abbildung 38 und Quellcodeabschnitt 6. In dem Verlauf des Roboterendeffektor ist erkennbar, dass ab ca. der Hälfte des Hohlzylinders, die letzte Achse des Roboters eine andere Konfiguration einnimmt. Da für die Kollisionsgeometrien bisher noch die CAD-Dateien des Roboters genutzt werden und der TCP des Endeffektors außerhalb der CAD-Geometrie liegt, ist der TCP-Verlauf teilweise innerhalb des Bauteils verschwunden. Die Rechendauer beträgt im Mittelwert  $\bar{x} = 2,181\text{ s}$  mit der Standardabweichung  $\sigma = 0,180\text{ s}$ .



**Abbildung 38:** Endeffektorbewegung einer inneren Kehlnaht aus mehreren Schweißnähten

**Quellcode 6:** Codeausschnitt der inneren Kehlnaht aus mehreren Schweißnähten

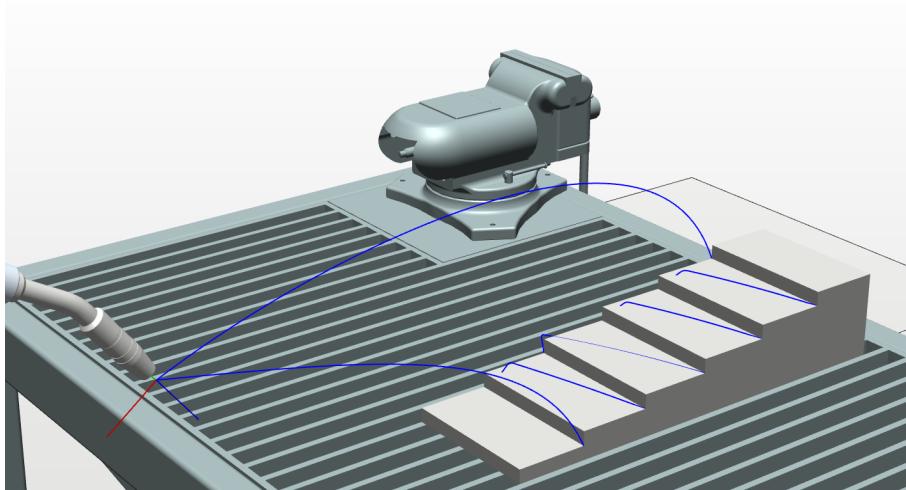
```

1 useMilliMeters();
2 useDegree();
3 setCoordinationSystem(Pose{1000, 0, 710, 0, 0, 1, 0});
4 val welds = 10;
5 val halfAngle = 360.0/(2*welds);
6 val quaterAngle = 360.0/(4*welds);

8 // sin and cos use rad
9 start(z(-cos(rad(30)), euler(0, 30, 0));
10 start(addEuler(0, 0, -halfAngle));
11 for(var i=0; i<welds; ++i) {
12     val startAngle = i*2*M_PI/welds;
13     val viaAngle = startAngle + rad(quaterAngle);
14     val endAngle = startAngle + rad(halfAngle);
15     start(x(cos(startAngle)*295), y(sin(startAngle)*295), addEuler(0, 0, halfAngle));
16     task(turnOn);
17     moveCvia(x(cos(viaAngle)*295), y(sin(viaAngle)*295), addEuler(0, 0, quaterAngle));
18     moveCto(x(cos(endAngle)*295), y(sin(endAngle)*295), addEuler(0, 0, quaterAngle));
19     task(turnOff);
20     if(i<welds-1) {
21         nextPath();
22     }
23 }

```

Das letzte einfache Schweißbeispiel besteht in dem Schweißen mehrerer Kehlnähte. Mittelwert und Standardabweichung der Berechnungsdauer betragen:  $\bar{x} = 1,327\text{ s}$  und  $\sigma = 0,215\text{ s}$ .



**Abbildung 39:** Endeffektorbewegung mehrerer Kehlnähte

**Quellcode 7:** Codeausschnitt der mehreren Kehlnähte

```

1 useMilliMeters();
2 useDegree();
3 setCoordinationSystem(Pose{800, 0, 700, 0, 0, 1, 0});
4 start(euler(0,-45,180));
5 for(var i=0; i<5; ++i) {
6     start(addX(-100), y(0), addZ(-20));
7     task(turnOn);

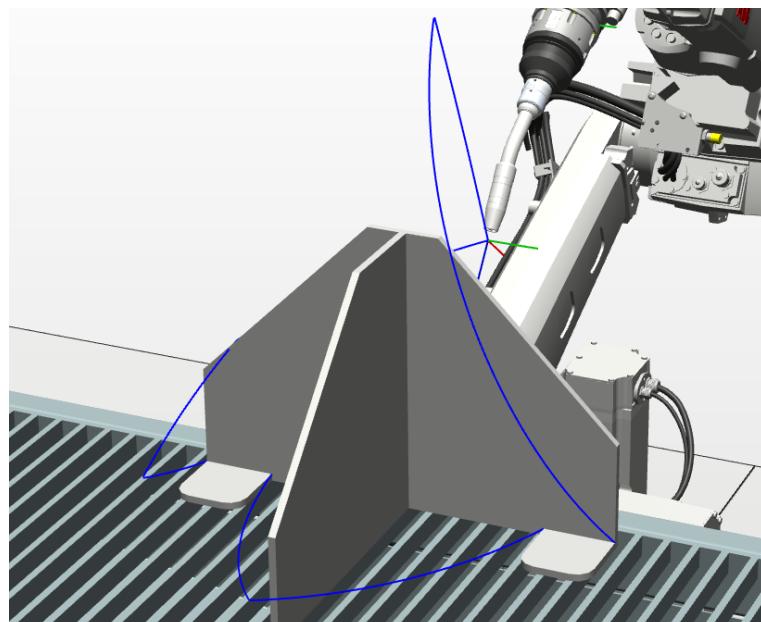
```

```
8     moveL(y(200));
9     task(turnOff);
10    if(i<4)
11        nextPath();
12 }
```

## 9.2 Barrieren

In den folgenden zwei Schweißbeispielen wurden die Wege zum Erreichen der Prozessbahnen durch Bleche erschwert.

In dem ersten der beiden besteht der Prozess aus zwei Kehlnähten. Zwischen diesen muss der Roboter ohne Kollision um die Bleche navigieren, siehe Abbildung 40. Der Mittelwert der Berechnungsduer dieses Beispiels beträgt  $\bar{x} = 9,667\text{ s}$ , die Standardabweichung  $\sigma = 1,242\text{ s}$ .



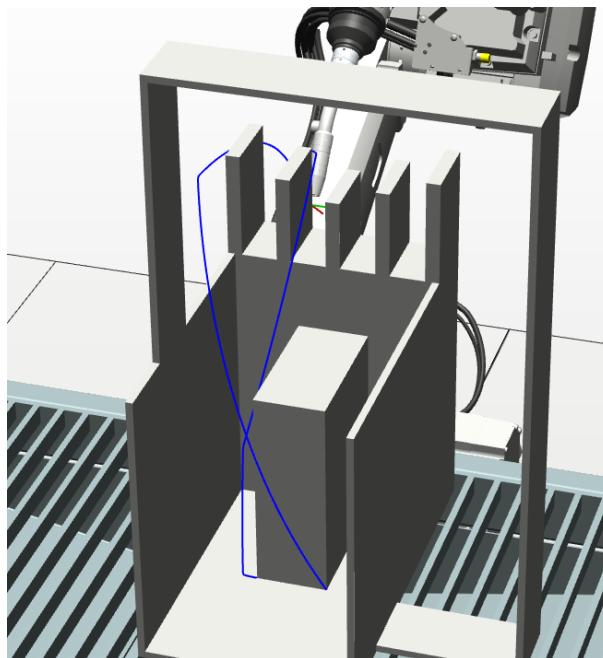
**Abbildung 40:** Endeffektorbewegung zweier Kehlnähte mit Barriere

### Quellcode 8: Codeausschnitt der Barriere mit zwei Kehlnähten

```
1 useMilliMeters();
2 useDegree();
3 setCoordinationSystem(Pose{600, -300, 710, 0, 0, 1, 0});
4 start(x(-10), euler(0,45,0));
5 task(turnOn);
6 moveL(addY(100));
7 task(turnOff);

9 nextPath();
10 start(y(500));
11 task(turnOn);
12 moveL(addY(100));
13 task(turnOff);
```

Das letzte Beispiel besteht aus einer komplizierten Schweißkonstruktion und einer einzigen Kehlnaht. Der Roboter darf auf dem Weg dorthin nicht mit den Blechen kollidieren und muss durch sie hindurch navigieren, siehe Abbildung 41. Mittelwert und Standardabweichung betragen:  $\bar{x} = 32,360\text{ s}$  und  $\sigma = 18,874\text{ s}$ .



**Abbildung 41:** Endeffektorbewegung einer Kehlnaht mit komplizierter Barriere

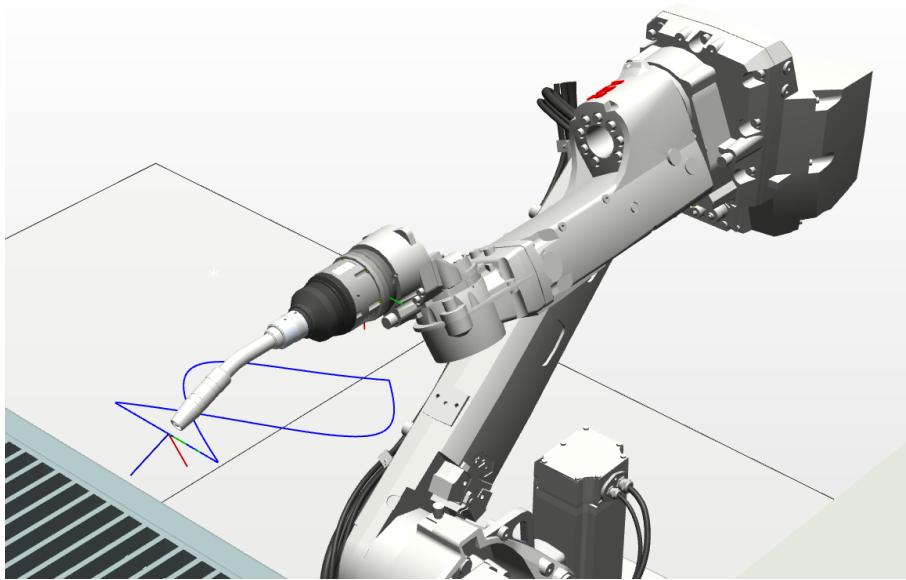
**Quellcode 9:** Codeausschnitt der Kehlnaht mit komplizierter Barriere

```
1 useMilliMeters();
2 useDegree();
3 setCoordinationSystem(Pose{600, 0, 710, 0, 0, 1, 0});
4 start(x(-300), y(100), euler(0, 45, 0));
5 task(turnOn);
6 moveL(addY(100));
7 task(turnOff);
```

### 9.3 Singularität

Keines der obigen Schweißbeispiele hat eine Singularität erzeugt. Das liegt unter anderem daran, dass Schweißbrenner für Industrieroboter eine Krümmung besitzen. Durch diese werden bereits viele mögliche Singularitäten mechanisch vermieden, da die vorletzte Achse so seltener auf  $0^\circ$  steht.

Abbildung 42 zeigt eine lineare Bahn, welche in der Mitte genau diese Singularität zwischen Achse 3 und Achse 5 erzeugt.



**Abbildung 42:** Endeffektorbewegung durch eine Singularität

Wird die Bahn mithilfe der passenden Wegpunktbefehle generiert, in ABB RAPID: MoveL, stoppt die Bewegung des ABB Roboters vor der Ausführung. Das liegt daran, dass der Roboter seine Endeffektorgeschwindigkeit durch die Singularität nicht einhalten kann. An dieser Stelle muss Achse 3 die Konfiguration um ca.  $\pi$ , bzw.  $180^\circ$  ändern.

Bei der Bewegung des ABB Roboters mit Konfigurationsbefehlen, in ABB RAPID: MoveAbsJ, und einer Schrittweite von z. B.  $1\text{ mm}$ , wird die Bahn hingegen ausgeführt. Die Endeffektorgeschwindigkeit wird allerdings bei der Durchschreitung der Singularität nicht mehr eingehalten. Die Berechnung der maximalen Achsgeschwindigkeiten bei der Abtastung der Konfigurationen, reicht für diese Erkennung nicht aus (mehr zur Bestimmung naheliegender Konfigurationen, siehe Abschnitt 6.1). Das könnte daran liegen, dass die maximalen Achsbeschleunigungen nicht in die Berechnung einfließen.

Die Singularität aus Abbildung 42 wurde dadurch erzeugt, die vorletzte Achse auf  $0^\circ$  zu setzen. Wenn sie hingegen auf  $0,75^\circ$  gesetzt wird, schafft es der ABB Roboter wieder mit dem linearem Wegbefehl. Bei der Bewegung kann die Endeffektorgeschwindigkeit eingehalten werden, da der Konfigurationswechsel so ein wenig Zeit erhält und nicht auf der Stelle ausgeführt werden muss.

Um die maximal mögliche Endeffektorgeschwindigkeit mit der entwickelten Roboterbibliothek zu ermitteln, wurde ein Programm erzeugt, dass bei  $50\text{ mm/s}$  anfängt und solange die Geschwindigkeit reduziert, bis die Berechnung erfolgreich verlaufen ist. Wie dieses Ergebnis zeigt, wird bei einem Prozess bei dem die Einhaltung der Endeffektorgeschwindigkeit notwendig ist, wie dem Schweißen, eine Erweiterung der Berechnung mithilfe der maximalen Achsbeschleunigungen benötigt.

## 10 Diskussion

In diesem Kapitel geht es um die Auswertung, der mit der entwickelten Roboterbibliothek entstandenen Ergebnisse. Dabei wird mit der gegebenen Zielsetzung verglichen, inwieweit die Ausarbeitung einsetzbar ist. Abschließend wird ein Ausblick auf weitere Schritte gegeben, wie die Roboterbibliothek erweitert werden könnte. Ziel dieser Arbeit war es, einen Industrieroboter mithilfe zuvor generierter Bahnen, kollisionsfrei über ein Bauteil zu manövrieren.

### 10.1 Zusammenfassung

Hintergrund der Entwicklung ist die Ansteuerung eines ABB IRB 2600ID Industrieroboters für ein Forschungsprojekt zur automatischen Dekontamination unbekannter Bauteile. Um die Software auch für andere Projekte nutzbar zu machen, wurde eine Roboterbibliothek entworfen.

Diese nutzt für die inverse Kinematik die analytische Methode von OpenRAVE's IKFast, um einen höheren Abdeckungsgrad durch die Mehrdeutigkeiten zu erlangen. Mithilfe von OpenRAVE ließ sich ebenfalls die Kollision mit der Bibliothek FCL testen. Somit wird jede gefundene Konfiguration auf Kollision geprüft.

Das Kernelement der entwickelten Roboterbibliothek ist die Bahnplanung. Sie ermöglicht die Eingabe von Wegpunkten, um einzelne Prozessbahnen berechnen zu können. Durch die Trennung der Prozesse, wie einer Schweißnaht, mit den Wegen zwischen diesen, wird die Programmierung leichter und besitzt bei verschiedenen unbekannten Bauteilen, einen höheren Abdeckungsgrad. Für die einheitlichen Wegpunkte sind bereits zwei Implementierungen umgesetzt worden, nämlich lineare und zirkulare.

Nachdem die einzelnen Prozessbahnen berechnet sind, d. h. für die Bahnen wurden die einzelnen Konfigurationen erfolgreich, ohne Kollision berechnet, werden die Zwischenwege ermittelt. Zwischen den Prozessen hat die Bahn oft keine Bedingungen, abgesehen davon, dass sie keine Kollision verursachen soll. Da die direkte *Point to Point* Bewegung zwischen den Konfigurationen oft nicht ohne Kollision möglich ist, verwendet die entwickelte Roboterbibliothek den *RRT-Connect*-Algorithmus. Diesen Implementierung breitet kollisionsfreie Wege der sechs Achsen, von der Start- und Zielkonfiguration, mit Baumstrukturen aus. Sobald eine Verbindung besteht, wird dieser Weg durch Abkürzungen vereinfacht.

Um den Robotor anzusteuern, lassen sich in den Wegpunkten noch Geschwindigkeits- und Zonenoptionen angeben. Für weitere notwendige Aufgaben, wie dem Einschalten eines Werkzeugs, lassen sich Roboteraufgaben mit dem passendem Robotercode definieren.

Sobald die komplette Trajektorie, bestehend aus den Prozessbahnen und den Zwischenwegen, berechnet wurde, wird diese in einer Visualisierung angezeigt. Außerdem wird mithilfe des Postprozessors gültiger Roboterprogrammcode generiert, oder der Roboter wird zur Laufzeit mit Befehlen angesprochen, um die Trajektorie abzufahren.

Das letzte entwickelte Modul der Roboterbibliothek ist die Eingabeschnittstelle. Diese vereinfacht die Eingabe von Prozessbahnen und ist, besonders durch die Python-Anbindung, auch für die normale, manuelle Roboterprogrammierung verwendbar.

## 10.2 Auswertung

Wie die Ergebnisse in Kapitel 9 zeigen, ist es mit der entwickelten Roboterbibliothek auf einfache Weise möglich, einen Industrieroboter kollisionsfrei zu programmieren. Durch die Umsetzung der Software in der Programmiersprache C++, lassen sich generierte Prozessbahnen direkt überprüfen und ausführen.

In dem Forschungsprojekt zur automatischen Dekontamination unbekannter Bauteile, sollen beispielsweise Bahnen aus Punktfolgen generiert werden. Da für die 3D-Bildverarbeitung ebenfalls C++ verwendet wird, ist eine Schnittstelle zwischen den Programmteilen leicht umzusetzen. Sobald eine Bahn zur Kollision führt, kann somit eine Stellung des Drehtisches berechnet werden, an dem die Bahn eventuell doch durchführbar ist.

Für viele Prozesse ist die Rechendauer der Software nicht relevant. So auch bis zu einem gewissen Grad in dem erwähntem Forschungsprojekt. Wie in den Ergebnissen erkennbar, hängt die Rechenzeit hauptsächlich von dem Bestimmen des Konfigurationsweges ab. Das erste Beispiel mit der langen V-Naht, benötigt z. B. im Schnitt nur  $\bar{x} = 0,540\text{ s}$ . Im Gegensatz dazu benötigt die Berechnung der Barriere, bestehend aus zwei Kehlnähten, im Schnitt bereits  $\bar{x} = 9,667\text{ s}$ . Das liegt daran, dass die Zwischenwege bei der langen V-Naht bereits mit einer direkten *PtP*-Bewegung möglich sind, hingegen die Barriere den *RRT-Connect*-Algorithmus zur Berechnung des Konfigurationsweges benötigt.

Für zeitkritische Anwendungen empfiehlt es sich, die genutzten Einstellungen zur Ausführung des *RRT-Connect*-Algorithmus anzupassen, um somit die Rechenzeit zu kürzen. Mit dem passendem Kollisionsmodell des Roboters, indem auch eine Toleranz einbezogen wurde, kann z. B. die Schrittweite des Kollisionstests vergrößert werden. Außerdem verringert sich, durch Erhöhung der eigentlichen Schrittgröße  $\epsilon$ , die Anzahl an Knotenpunkten in der Baumstruktur. Beide Schrittgrößen sorgen hingegen dafür, dass der Roboter weniger Spielraum bei der Bewegung erhält und somit komplizierte Wege eventuell nicht erfolgreich berechnet werden können.

Die Einstellungen sind also den Anwendungsbedingungen anzupassen.

Wie die Quellcodeabschnitte der Ergebnisse zeigen, ist mithilfe der Eingabeschnittstelle, eine einfache Beschreibung der Prozessbahnen möglich. So kann die entwickelte Roboterbibliothek auch für andere Projekte genutzt werden.

Abschnitt 9.3 der Ergebnisse zeigt den nicht genügenden Umgang mit Singularitäten innerhalb der Software. Da u. a. die maximalen Achsbeschleunigungen nicht mit in die Berechnung einfließen, kann nicht für eine erfolgreiche Ausführung garantiert werden. Bei der Ansteuerung des Roboters mit den passenden Wegbefehlen, stoppt beispielsweise die ABB-Steuerung vor der eintretenden Singularität. Werden hingegen für Wegpunkte, Zwischenposen mit einer *PtP*-Bewegung verwendet, wird die Endeffektorgeschwindigkeit nicht eingehalten. Die Endeffektorbewegung gerät dann bei Eintritt der Singularität kurz zum Stehen, um die Konfiguration anzupassen.

Die Ergebnisse zeigen, dass die entwickelte Roboterbibliothek dieser Arbeit, zur kollisionsfreien Roboterprogrammierung eingesetzt werden kann. Für das Forschungsprojekt, zur automatischen Dekontamination unbekannter Bauteile, ist die nicht eingehaltene Endeffektorgeschwindigkeit bei eintretender Singularität nicht relevant. Soll hingegen beispielsweise mit der Software geschweißt werden, sollte sie bereits Singularitäten während der Berechnung erkennen können.

### 10.3 Ausblick

Die entwickelte Roboterbibliothek lässt sich noch in vielen Bereichen ergänzen. Im Folgenden werden einige wichtige und nützliche Erweiterungen vorgestellt:

Der wichtigste Punkt liegt in der korrekten Erkennung von Singularitäten. Wie bereits erwähnt, benötigt die Berechnung der Bahnplanung dazu mehr Informationen und eine dynamische Simulation. Dabei sollte die Software flexibel bleiben und auch die momentane Berechnung, ohne Einhaltung der Endeffektorgeschwindigkeit, nach Wunsch ermöglichen.

Falls der Roboter unerwartete Fehler erhält, könnte über eine Fehlerrückmeldung, statt einem Programmstopp, auf diese in der Software eingegangen werden. Darunter fällt auch der Stopp wegen dem Singularitätsfehler. Für die Rückmeldung kann die vorhandene TCP-Verbindung zum Roboter genutzt werden, da die Rückleitung bisher nicht verwendet wird. Weitere Signale, wie Trigger, können dann auch diese

Verbindung zur Verarbeitung innerhalb der Software nutzen.

Für Stabilität der Roboterbibliothek sollten an einigen Stellen weitere Fehlerbehandlungen implementiert werden, wie dem nicht erfolgreichen Verarbeiten von Einstellungen. Für eine nachträgliche Fehleranalyse fehlt außerdem noch ein Protokollsystem, welches Außergewöhnlichkeiten in einer Datei protokolliert.

Die Darstellung des Roboters und der berechneten Trajektorie, wird zur Zeit mit der Visualisierung aus OpenRAVE umgesetzt. Für den industriellen Einsatz wird eine einbettbare Visualisierung benötigt, um dem Programm spezielle Eigenschaften, wie Knöpfe und Statusfelder, hinzufügen zu können.

Der entwickelten Roboterbibliothek fehlt noch eine Anbindung zur virtuellen Umgebung in der Laufzeit. Die CAD-Dateien und dessen Posen werden bisher nur aus der OpenRAVE XML-Datei geladen. Das Hinzufügen, Transformieren und Entfernen dieser, ist zur Laufzeit wünschenswert. Des Weiteren könnten primitive Geometrien, wie Sphären, Zylinder und Quader, für einige Anwendungen hilfreich sein.

Während der Verfassung dieser Masterarbeit wurde aus Zeitgründen auf das Erstellen eines Kollisionsmodells des Roboters verzichtet. Dieses ist für die Berechnung zwingend erforderlich, damit auf Toleranzen bei Bewegungen eingegangen wird. Um die Erstellung solch eines Kollisionsmodells zu vereinfachen, könnte ein separates Programm entwickelt werden, welches den Nutzer bei der Generierung unterstützt. Beispielsweise durch Vergrößern der CAD-Dateien um einen definierten Abstand.

Ändert sich der Prozess während der Bearbeitung durch den Roboter, kann um eine Live-Kollisionskontrolle ergänzt werden. Hierzu schickt der Roboter permanent seine aktuelle Ist-Konfiguration zur Software zurück. Diese kann dann kontrollieren, ob er in der virtuellen Umgebung eine Kollision verursachen könnte. Wird eine 3D-Kamera hinzugefügt, sind auch Änderungen der Umgebung bemerkbar. Für die Verbindung sollte das UDP-Protokoll, statt dem TCP-Protokoll, verwendet werden. Dessen Verbindungsauflaufbau ist unanfällig für Fehler und das Verschlucken einer Nachricht ist nicht kritisch. Wichtiger ist die Aktualität der Nachrichten.

Um die Rechendauer zu verkürzen, lässt sich die Bibliothek um Multitasking ergänzen, indem so mehrere Wegpunkte und Bahnen parallel berechnet werden. Eine weitere Möglichkeit besteht in dem Optimieren des *RRT-Connect*-Algorithmus, da dieser einen Großteil der Rechenzeit einnimmt.

Einige Anwendungen benötigen mehrere Roboter in der gleichen Bearbeitungszelle. Eine Erweiterung um die simultane kollisionsfreie Bewegung mehrerer Roboter ist möglich, aber aufwendig. Leichter ist die Umsetzung, wenn sich nur einer dieser Roboter bewegt, während die anderen stehen bleiben. In diesem Fall können stehende Roboter wie andere Bauteile in der virtuellen Umgebung betrachtet werden.

*Pick and Place* Anwendungen benötigen oft keine direkte Bahn. Die vorhandene Berechnung des Konfigurationsweges genügt hierfür nicht immer aus, da das Bauteil z. B. nicht immer kippen darf. Hierzu wird ein neuer Algorithmus benötigt, der auch mit Endeffektoreinschränkungen umgehen kann. Dieser könnte auch in Zusammenhang mit einem numerischen Verfahren der inversen Kinematik umgesetzt werden.

Für die Bahnplanung wurden bisher zwei Wegpunktarten umgesetzt, die der linearen und zirkularen Wegpunkte. Weitere Wegpunktarten wie Spline-Wegpunkte könnten in manchen Anwendungen hilfreich sein, um nicht stattdessen diese Bahn aus vielen linearen Wegpunkten erstellen zu müssen.

Der Live-Interpreter von Python hilft bereits bei der schnelleren Entwicklung eines Roboterprogrammes, mit dieser Roboterbibliothek. Durch Erweiterung dessen Funktionalität, könnte er den Roboter auch direkt mit Befehlen steuern, um ihn so beispielsweise  $1\text{ mm}$  in eine Richtung zu bewegen. So kann er auch für die Kalibrierung langsam in eine Richtung fahren, bis ein Trigger-Signal erscheint.

Die letzte erwähnenswerte Erweiterung liegt in dem Erstellen von Wegpunkten mit einer Toleranz der Posen. Einige Prozesse benötigen nicht die genaue Einhaltung der Bahnposen. Für diese würde die Toleranz der Position oder der Orientierung, z. B.  $\pm 10\text{ mm}$  und  $\pm 20^\circ$ , für mehr mögliche kollisionsfreie Bahnen sorgen. Für die Umsetzung könnte ein numerisches Verfahren der inversen Kinematik verwendet werden, da sich diese an das Ziel herantasten.

## Literatur

- [1] S. R. Buss *Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods*, Paper, University of California, San Diego, 2009,
- [2] S. Chiaverini, B. Siciliano, O. Egeland, *Review of the Damped Least-Squares Inverse Kinematics with Experiments on an Industrial Robot Manipulator*, Paper, 1994
- [3] J. Denavit, R. S. Hartenberg *A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices*, Paper, Journal of Applied Mechanics, 1955
- [4] R. Diankov *Automated Construction of Robotic Manipulation Programs*, Doktorarbeit, Carnegie Mellon University Pittsburgh, Pennsylvania, 2010
- [5] A.-V. Duka *Neural network based inverse kinematics solution for trajectory tracking of a robotic arm*, Paper, University of Tg. Mureş, Romania, 2014
- [6] A. A. Goldenberg, B. Benhabib, R. G. Fenton *A Complete Generalized Solution to the Inverse Kinematics of Robots*, Paper, 1985,
- [7] H. Holtkamp *Einführung in TCP/IP*, Seminarvortrag, Technische Fakultät, Universität Bielefeld, 2002
- [8] J. J. Kuffner Jr., S. M. LaValle *RRT-Connect: An Efficient Approach to Single-Query Path Planning*, Paper, Conf. on Robotics and Automation, USA, 2000
- [9] J.-M. Lien, N. M. Amato *Approximate Convex Decomposition of Polyhedra*, Paper, Solid and physical modeling, Beijing, China, 2007
- [10] D. Nauck, F. Klawonn, R. Kruse *Neuronale Netze und Fuzzy-Systeme, 2. Auflage*, Wiesbaden, Springer Verlag, 1996, ISBN: 978-3-528-15265-9
- [11] Z. Pan, J. Polden, N. Larkin, S. van Duin, J. Norrish *Recent progress on programming methods for industrial robots*, Paper, University of Wollongong, Australia, 2012
- [12] K. Shoemake *Animation Rotation with Quaternion Curves*, Paper, Siggraph, San Francisco, USA, 1985
- [13] W. Weber *Industrieroboter, Methoden der Steuerung und Regelung, 2. Auflage*, Darmstadt, Carl Hanser Verlag, 2009, ISBN: 978-3-446-41031-2
- [14] ABB *Überblick ABB-Industrieroboter*, Homepage, 2018, URL: <https://new.abb.com/products/robotics/de/industrieroboter/> (besucht am 08.06.2018)
- [15] ABB *Introduction to RAPID*, Handbuch, ABB AB Robotics Products, Schweden, 2007
- [16] ABB *RAPID Instructions, Functions and Data types* Referenzbuch, ABB AB Robotics Products, Schweden, 2010

- [17] Boost *Homepage des Pythonmoduls von Boost*, Homepage, 2018, URL: [https://www.boost.org/doc/libs/1\\_66\\_0/libs/python/doc/html/index.html](https://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/index.html) (besucht am 21.07.2018)
- [18] Bullet Physics Library *Homepage von Bullet*, Homepage, 2018, URL: <http://bulletphysics.org/wordpress/> (besucht am 16.06.2018)
- [19] Coppelia Robotics *Homepage von V-REP*, Homepage, 2018, URL: <http://www.coppeliarobotics.com/index.html> (besucht am 16.06.2018)
- [20] DIN EN ISO *Roboter und Robotikgeräte - Wörterbuch*, Norm, 2010, DIN EN ISO 8373:2010
- [21] Doxygen *Homepage der Dokumentationsgenerierungssoftware Doxygen*, Homepage, 2018, URL: <http://www.stack.nl/~dimitri/doxygen/> (besucht am 20.07.2018)
- [22] Eigen *Homepage der C++ Bibliothek Eigen*, Homepage, 2018, URL: [http://eigen.tuxfamily.org/index.php?title=Main\\_Page](http://eigen.tuxfamily.org/index.php?title=Main_Page) (besucht am 25.06.2018)
- [23] Flexible Collision Library *Homepage von FCL*, Homepage, 2018, URL: <https://github.com/flexible-collision-library/fcl> (besucht am 16.06.2018)
- [24] Flexible Collision Library *Vergleich verschiedener Kollisionsbibliotheken*, Homepage, 2018, URL: [https://github.com/personalrobotics/or\\_fcl](https://github.com/personalrobotics/or_fcl) (besucht am 16.06.2018)
- [25] IFR *Roboter bis 2020 weltweit verdoppelt*, Pressemitteilung, 2018, URL: [https://ifr.org/downloads/press2018/2018-Mai-30\\_Pressemitteilung\\_IFR\\_Roboter\\_Workplace\\_deutsch.pdf](https://ifr.org/downloads/press2018/2018-Mai-30_Pressemitteilung_IFR_Roboter_Workplace_deutsch.pdf) (besucht am 8.06.2018)
- [26] MoveIt! *Homepage von MoveIt!*, Homepage, 2018, URL: <https://moveit.ros.org/documentation/concepts/> (besucht am 30.05.2018)
- [27] MoveIt! *Quellcode der inversen Kinematik von MoveIt!*, Quellcode, 2018, URL: [https://github.com/ros-planning/moveit/blob/kinetic-devel/moveit\\_core/kinematics\\_base/src/kinematics\\_base.cpp](https://github.com/ros-planning/moveit/blob/kinetic-devel/moveit_core/kinematics_base/src/kinematics_base.cpp) (besucht am 1.06.2018)
- [28] MoveIt! *Quellcode der IKFast Erweiterung für MoveIt!*, Quellcode, 2018, URL: [https://github.com/ros-planning/moveit/blob/kinetic-devel/moveit\\_kinematics/ikfast\\_kinematics\\_plugin/templates/ikfast61\\_moveit\\_plugin\\_template.cpp](https://github.com/ros-planning/moveit/blob/kinetic-devel/moveit_kinematics/ikfast_kinematics_plugin/templates/ikfast61_moveit_plugin_template.cpp) (besucht am 1.06.2018)
- [29] MoveIt! *Quellcode der Bewegungsplanung von MoveIt!*, Quellcode, 2018, URL: [https://github.com/ros-planning/moveit/blob/kinetic-devel/moveit\\_ros\\_planning\\_interface/move\\_group\\_interface/src/move\\_group\\_interface.cpp](https://github.com/ros-planning/moveit/blob/kinetic-devel/moveit_ros_planning_interface/move_group_interface/src/move_group_interface.cpp) (besucht am 1.06.2018)
- [30] MoveIt! *Quellcode des ABB RAPID Roboterprogramms von MoveIt!*, Quellcode, 2018, URL: [https://github.com/ros-industrial/abb/tree/kinetic-devel/abb\\_driver/rapid](https://github.com/ros-industrial/abb/tree/kinetic-devel/abb_driver/rapid) (besucht am 11.07.2018)

- [31] Open Dynamics Engine *Homepage von ODE*, Homepage, 2018, URL: <http://www.ode.org/> (besucht am 16.06.2018)
- [32] OpenRAVE *Homepage der OpenRAVE*, Homepage, 2018, URL: [www.openrave.org/](http://openrave.org/) (besucht am 22.05.2018)
- [33] OpenRAVE *Dokumentation des OpenRAVE XML-Formats*, Dokumentation, 2018, URL: <http://openrave.programmingvision.com/wiki/index.php/Format:XML> (besucht am 6.06.2018)
- [34] Orocus KDL *Dokumentation von Orocus KDL*, Dokumentation, 2018, URL: [http://docs.ros.org/indigo/api/orocos\\_kdl/html/classKDL\\_1\\_1ChainIkSolverPos\\_\\_LMA.html](http://docs.ros.org/indigo/api/orocos_kdl/html/classKDL_1_1ChainIkSolverPos__LMA.html) (besucht am 1.06.2018)
- [35] Python *Homepage von Python*, Homepage, 2018, URL: <https://www.python.org/> (besucht am 21.07.2018)
- [36] RoboDK *Homepage der RoboDK*, Homepage, 2018, URL: <https://robodk.com/> (besucht am 22.05.2018)
- [37] Robotics Library *Homepage der Robotics Library*, Homepage, 2018, URL: <https://www. roboticslibrary.org/> (besucht am 22.05.2018)
- [38] ROS-Industrial *Homepage von ROS-Industrial*, Homepage, 2018, URL: <https://rosindustrial.org/about/description/> (besucht am 30.05.2018)
- [39] ROS-Industrial *Kurzanleitungen von ROS-Industrial*, Homepage, 2018, URL: <http://wiki.ros.org/Industrial/Tutorials> (besucht am 30.05.2018)

## A Konfigurationsdateien für OpenRAVE

Die folgenden XML-Dateien beschreiben die Roboterkinematik des ABB IRB 2600ID mit einem Schweißwerkzeug.

Mehr zum OpenRAVE XML-Format, siehe [33]

**Quellcode 10:** OpenRave-Umgebungs-XML-Datei eines einzelnen ABB IRB 2600ID

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <!-- Filename: env.single.robot.xml -->
3  <Environment>
4      <Robot file="abbirb2600id_binzel.robot.xml" />
5  </Environment>
```

**Quellcode 11:** OpenRave-Umgebungs-XML-Datei des ABB IRB 2600ID in einer Schweißumgebung

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <!-- Filename: env.robot.xml -->
3  <Environment>
4      <Robot file="abbirb2600id_binzel.robot.xml" />

6      <Kinbody name="cabinet" >
7          <Body name="binzelbody" >
8              <Translation>
9                  0.0  1.0  0.1
10             </Translation>
11             <quat>
12                 0 0 0.70711 0.70711
13             </quat>
14             <Geom type="trimesh" >
15                 <Data>addcad/controller.wrl  1</Data>
16                 <Render>addcad/controller.wrl  1</Render>
17             </Geom>
18         </Body>
19     </Kinbody>

21     <Kinbody name="workbench" >
22         <Body name="binzelbody" >
23             <Translation>
24                 1.0  0.0  -0.1
25             </Translation>
26             <RotationMat>
27                 1.0  0.0  0.0
28                 0.0  1.0  0.0
29                 0.0  0.0  1.0
30             </RotationMat>
31             <Geom type="trimesh" >
32                 <Data>addcad/weldingbench.wrl  0.001</Data>
33                 <Render>addcad/weldingbench.wrl  0.001</Render>
34             </Geom>
35         </Body>
36     </Kinbody>

38     <Kinbody name="weldpart" >
39         <Body name="binzelbody" >
40             <Translation>
```

```

41           0.7  0.0  0.7
42       </Translation>
43     <RotationMat>
44       1.0  0.0  0.0
45       0.0  1.0  0.0
46       0.0  0.0  1.0
47   </RotationMat>
48   <Geom type="trimesh">
49     <Data>addcad/weldpart1.wrl  0.001</Data>
50     <Render>addcad/weldpart1.wrl  0.001</Render>
51   </Geom>
52 </Body>
53 </Kinbody>
54 </Environment>

```

**Quellcode 12:** OpenRave-Roboter-XML-Datei des ABB IRB 2600ID mit Schweißwerkzeug

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <!-- Filename: abbirb2600id_binzel.robot.xml -->
3  <Robot name="abbirb2600id_binzel">
4    <KinBody file="abbirb2600id.kinbody.xml">
5      <Body name="binzelbody">
6        <offsetfrom>Arm5</offsetfrom>
7        <Translation>
8          0.049841566361  0.0  0.372194826196
9        </Translation>
10       <RotationMat>
11         0.9271839  0.0  0.3746066
12         0.0        1.0  0.0
13         -0.3746066 0.0  0.927183
14       </RotationMat>
15       <Geom type="trimesh">
16         <Translation>
17           0.09321433  0.0  -0.363764
18         </Translation>
19         <RotationMat>
20           0.9271839  0.0  -0.3746066
21           0.0        1.0  0.0
22           0.3746066  0.0  0.9271839
23         </RotationMat>
24         <Data>cad/Binzel.wrl  1.0</Data>
25         <Render>cad/Binzel.wrl  1.0</Render>
26       </Geom>
27     </Body>

28     <Joint type="hinge" name="binzeljoint" enable="false">
29       <body>Arm5</body>
30       <body>binzelbody</body>
31       <limits>0 0</limits>
32     </Joint>
33   </KinBody>
34   <Manipulator name="binzel">
35     <effector>binzelbody</effector>
36     <base>Base</base>
37   </Manipulator>
38 </Robot>

```

**Quellcode 13:** OpenRave-Kinematik-XML-Datei des ABB IRB 2600ID

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <!-- Filename: abbirb2600id.kinbody.xml -->
3  <KinBody name="ABBIRB2600IF185kinbody">
4      <Body name="Base" type="dynamic">
5          <Translation>0.0 0.0 0.0</Translation>
6          <RotationMat>1 0 0 0 1 0 0 0 1</RotationMat>
7          <Geom type="trimesh">
8              <Translation>0.0 0.0 0.0</Translation>
9              <rotationaxis>0.0 0.0 1.0 0</rotationaxis>
10             <Data>cad/Base.wrl 1.0</Data>
11             <Render>cad/Base.wrl 1.0</Render>
12         </Geom>
13     </Body>

15     <Body name="Arm0" type="dynamic">
16         <Translation>0.0 0.0 0.0</Translation>
17         <RotationMat>1 0 0 0 1 0 0 0 1</RotationMat>
18         <Geom type="trimesh">
19             <Translation>0.0 0.0 0.0</Translation>
20             <RotationMat>1 0 0 0 1 0 0 0 1</RotationMat>
21             <Data>cad/Link1.wrl 1.0</Data>
22             <Render>cad/Link1.wrl 1.0</Render>
23         </Geom>
24     </Body>
25     <Joint circular="true" name="Joint0" type="hinge">
26         <Body>Base</Body>
27         <Body>Arm0</Body>
28         <offsetfrom>Arm0</offsetfrom>
29         <axis>0 0 1</axis>
30     </Joint>

32     <Body name="Arm1" type="dynamic">
33         <Translation>0.150 0.0 0.445</Translation>
34         <RotationMat>1 0 0 0 0 1 0 -1 0</RotationMat>
35         <Geom type="trimesh">
36             <Translation>-0.15 0.445 0</Translation>
37             <RotationMat>1 0 0 0 0 -1 0 1 0</RotationMat>
38             <Data>cad/Link2.wrl 1.0</Data>
39             <Render>cad/Link2.wrl 1.0</Render>
40         </Geom>
41     </Body>1
42     <Joint circular="true" name="Joint1" type="hinge">
43         <Body>Arm0</Body>
44         <Body>Arm1</Body>
45         <offsetfrom>Arm1</offsetfrom>
46         <axis>0 0 1</axis>
47     </Joint>

49     <Body name="Arm2" type="dynamic">
50         <Translation>0.150 0.0 1.345</Translation>
51         <RotationMat>1 0 0 0 0 1 0 -1 0</RotationMat>
52         <Geom type="trimesh">
53             <Translation>-0.15 1.345 0</Translation>
54             <RotationMat>1 0 0 0 0 -1 0 1 0</RotationMat>
55             <Data>cad/Link3.wrl 1.0</Data>
56             <Render>cad/Link3.wrl 1.0</Render>
57         </Geom>
58     </Body>

```

```
59      <Joint circular="true" name="Joint2" type="hinge">
60          <Body>Arm1</Body>
61          <Body>Arm2</Body>
62          <offsetfrom>Arm2</offsetfrom>
63          <axis>0 0 1</axis>
64      </Joint>

66      <Body name="Arm3" type="dynamic">
67          <Translation>0.0 0.0 1.495</Translation>
68          <RotationMat>0 0 1 0 1 0 -1 0 0</RotationMat>
69          <Geom type="trimesh">
70              <Translation>1.495 0 0</Translation>
71              <RotationMat>0 0 -1 0 1 0 1 0 0</RotationMat>
72              <Data>cad/Link4.wrl 1.0</Data>
73              <Render>cad/Link4.wrl 1.0</Render>
74          </Geom>
75      </Body>
76      <Joint circular="true" name="Joint3" type="hinge">
77          <Body>Arm2</Body>
78          <Body>Arm3</Body>
79          <offsetfrom>Arm3</offsetfrom>
80          <axis>0 0 1</axis>
81      </Joint>

83      <Body name="Arm4" type="dynamic">
84          <Translation>0.936 0.0 1.495</Translation>
85          <RotationMat>1 0 0 0 0 1 0 -1 0</RotationMat>
86          <Geom type="trimesh">
87              <Translation>-0.936 1.495 0</Translation>
88              <RotationMat>1 0 0 0 0 -1 0 1 0</RotationMat>
89              <Data>cad/Link5.wrl 1.0</Data>
90              <Render>cad/Link5.wrl 1.0</Render>
91          </Geom>
92      </Body>
93      <Joint circular="true" name="Joint4" type="hinge">
94          <Body>Arm3</Body>
95          <Body>Arm4</Body>
96          <offsetfrom>Arm4</offsetfrom>
97          <axis>0 0 1</axis>
98      </Joint>

100     <Body name="Arm5" type="dynamic">
101         <Translation>1.071 0.0 1.495</Translation>
102         <RotationMat>0 0 1 0 1 0 -1 0 0</RotationMat>
103         <Geom type="trimesh">
104             <Translation>1.495 0 -1.071</Translation>
105             <RotationMat>0 0 -1 0 1 0 1 0 0</RotationMat>
106             <Data>cad/Link6.wrl 1.0</Data>
107             <Render>cad/Link6.wrl 1.0</Render>
108         </Geom>
109     </Body>
110     <Joint circular="true" name="Joint5" type="hinge">
111         <Body>Arm4</Body>
112         <Body>Arm5</Body>
113         <offsetfrom>Arm5</offsetfrom>
114         <axis>0 0 1</axis>
115     </Joint>
116 </KinBody>
```

## B C++-Programm zum Testen von IKFast

Mithilfe des folgenden Programmes kann die von IKFast erzeugte C++-Quellcodedatei getestet werden:

**Quellcode 14:** IKFast-Testprogramm

```

1 #include <iostream>
2 #include <chrono>
3 #include <random>
4 #include <vector>
5 #include <array>

7 #include "ikfast.h"

9 #define var auto
10 #define val const auto
11 #define ref auto&&
12 #define viu const auto&

14 using namespace std;
15 using namespace std::chrono;

17 int main() {
18     // Random number generator
19     var random = default_random_engine{};
20     var revolutionDistribution = uniform_real_distribution<double>{
21         -M_PI, M_PI};

23     val calculations = 100000000;

25     // Results
26     var timings = vector<nanoseconds>(calculations);
27     var ikFailed = 0;
28     var solFailed = 0;
29     var freeCnt = 0;
30     var solCnt = vector<int>{};

32     for (var i = 0; i < calculations; ++i) {
33         if(i%1000==0)
34             cout << "Calculation #" << i << endl;

36         // Random joint values
37         var joints = array<double, 6>{};
38         for (ref joint : joints) {
39             joint = revolutionDistribution(random);
40         }

42         // Forward kinematic
43         var rotation = array<double, 9>{};
44         var translation = array<double, 3>{};
45         ik::ComputeFk(
46             joints._M_elems, translation._M_elems, rotation._M_elems);

49         // Inverse kinematic
50         val startTime = system_clock::now();
51         var solutions = ikfast::IkSolutionList<double>{};
```

```

52         val success = ik::ComputeIk(translation._M_elems, rotation._M_elems,
53                                     nullptr, solutions);
54         timings[i] = duration_cast<nanoseconds>(
55             system_clock::now() - startTime);
56
57         if(!success) {
58             ++ikFailed;
59             continue;
60         }
61
62         if(solutions.GetNumSolutions() > solCnt.size()) {
63             solCnt.resize(solutions.GetNumSolutions());
64         }
65
66         ++solCnt[solutions.GetNumSolutions()-1];
67
68         for(var s = 0; s < solutions.GetNumSolutions(); ++s) {
69             const ikfast::IkSolutionBase<double>& sol =
70                 solutions.GetSolution(s);
71
72             var solJoints = vector<double>(6);
73             var solFrees = vector<double>(sol.GetFree().size());
74             sol.GetSolution(solJoints, solFrees);
75
76             if(sol.GetFree().size()>0)
77                 ++freeCnt;
78
79             var solRotation = array<double, 9>{};
80             var solTranslation = array<double, 3>{};
81             ik::ComputeFk(solJoints.data(),
82                           solTranslation.data(), solRotation.data());
83
84             // Check if this solutionn is in the following range
85             var inRange = true;
86             val transTolerance = 0.01;
87             val rotTolerance = 0.002;
88
89             for(var t=0; t<3; ++t) {
90                 if(solTranslation[t] > translation[t]+transTolerance ||
91                     solTranslation[t] < translation[t]-transTolerance) {
92                     inRange = false;
93                     break;
94                 }
95             }
96             for(var r=0; r<9; ++r) {
97                 if(solRotation[r] > rotation[r]+rotTolerance ||
98                     solRotation[r] < rotation[r]-rotTolerance) {
99                     inRange = false;
100                    break;
101                }
102            }
103            if(!inRange)
104                ++solFailed;
105        }
106    }
107
108    // Calculate average time for the inverse kinematics
109    var averageTime = accumulate(
110        timings.begin(), timings.end(), nanoseconds{})

```

```

111             / timings.size();

113     // Show the results
114     cout << "\n\n\nRandom calculations #" << calculations << '\n';
115     cout << "Failed inverse kinematics #" << ikFailed << '\n';
116     cout << "Failed solution #" << solFailed << '\n';
117     cout << "Found singularities #" << freeCnt << '\n';
118     cout << "Average time for inverse kinematics: " <<
119         averageTime.count() << "ns\n";
120     for(var sols = 0; sols < solCnt.size(); ++sols) {
121         cout << "Found " << sols << " solutions #" << solCnt[sols] << '\n';
122     }
123     cout << endl;
124     return 0;
125 }

127 //Random calculations #100000000
128 //Failed inverse kinematics #78
129 //Failed solution #0
130 //Found singularities #90
131 //Average time for inverse kinematics: 12306ns
132 //Found 0 solutions #0
133 //Found 1 solutions #8
134 //Found 2 solutions #8
135 //Found 3 solutions #16275604
136 //Found 4 solutions #0
137 //Found 5 solutions #31
138 //Found 6 solutions #22
139 //Found 7 solutions #83724249

```

## C Testprogramm der Roboterschnittstelle

Mithilfe des folgenden C++-Programmes wurde der Roboter für das Schweißbauteilbeispiel programmiert. Aus dem Programm wird gültiger RAPID-Robotercode generiert und der Roboter wird zur Laufzeit gesteuert.

**Quellcode 15:** C++-Programmbeispiel des Schweißbauteils

```

1 #include <iostream>
2 #include <vector>
3 #include <thread>

5 #include "pose.h"
6 #include "kinematic.h"
7 #include "waypoint.h"
8 #include "configurationpath.h"
9 #include "pathplanning.h"
10 #include "rave/ikfastsolver.h"
11 #include "rave/raveplugin.h"
12 #include "rapidabb/rapidprocessor.h"
13 #include "rapidabb/rapidconnection.h"

15 #define var auto
16 #define val const auto
17 #define ref auto&&

```

```
18 #define viu const auto&
19
20 using namespace pf;
21 using namespace pf::rapid;
22 using namespace std;
23 using namespace Eigen;
24
25 constexpr val runTime = true;
26
27 int main() {
28     // 
29     // Setup
30     //
31     IKFastSolver ikFastSolver{};
32     RaveCollisionVisualisation collisionVisualisation{"../env/env.robot.xml"};
33
34     Kinematic kinematics{ikFastSolver, collisionVisualisation};
35     kinematics.setLimitsInDegree({
36         {-180, 180},
37         {-95, 155},
38         {-180, 75},
39         {-175, 175},
40         {-120, 120},
41         {-400, 400}
42     });
43     kinematics.setMaxJointSpeedsInDegree({175, 175, 175, 360, 360, 500});
44
45     kinematics.setDefaultConfiguration(Configuration{0, 0, 0, 0, 0, 0});
46     kinematics.setJointVotes({1.0, 1.0, 1.0, 0.5, 0.5, 0.1});
47
48     ConfigurationPathPlanning configurationPathPlanning{kinematics};
49     PathPlanning pathPlanning{kinematics, configurationPathPlanning};
50
51     Configuration home{
52         rad(0), rad(-70), rad(40), rad(0), rad(60), rad(0)
53     };
54     pathPlanning.setHome(home);
55
56     Visualisation visualisation{kinematics};
57
58     RapidPostProcessor postProcessor{kinematics};
59     RapidRunTimeConnection connection{kinematics, "192.168.135.69", 12345};
60     if(runTime) {
61         cout << "connecting..." << endl;
62         connection.connect();
63         cout << "Connected" << endl;
64         connection.jointMovement(home, nullptr);
65         connection.start();
66     }
67     //
68     // END Setup
69     //
70
71     val speed = 0.05;
72     val zone = 0.005;
73     var speedZoneConditional = make_shared<SpeedZoneConditional>(speed, zone);
74
75     // Weldgun on / off tasks           "RunTime procedure", "PostProcessor code"
76     var turnOn = make_shared<RobotTask>(
```

```
77         "TurnOnWeldingGun",
78         "WaitTime \\InPos,0;\n! SetDO do1, 1;");
79     var turnOff = make_shared<RobotTask>(
80         "TurnOffWeldingGun",
81         "WaitTime \\InPos,0;\n! SetDO do1, 0;");
82
83     // Position      Orientation as Quaternion
84     // x, y, z,      w, i, j, k
85     Pose workObject{0.7, 0.0, 0.710, 0, 0, 1, 0};
86
87     //
88     // V-seam welding
89     //
90
91     // Position      Quaternion is default
92     // x, y, z      1, 0, 0, 0
93     Pose start{-0.05, 0.0, 0.0};
94
95     Pose goal{start};
96     goal.position[1] = 0.3;           // y
97
98     // Transforming start and goal to the weld part "workObject"
99     start.transformThis(workObject);
100    goal.transformThis(workObject);
101
102    // Creates a LinearWay::Ptr from start to goal
103    var linear = make_shared<LinearWay>(start, goal);
104
105    // Adds speed and zone
106    linear->executionConditional = speedZoneConditional;
107
108    // Adds start and end tasks
109    linear->startTask = turnOn;
110    linear->endTask = turnOff;
111
112    // Creates a Path with a single WayPoint
113    Path pathLinear{{linear}};
114
115    //
116    // Round fillet welding
117    // (needs 2 CircularWays, because a CircularWay must be < 360 degrees
118    //
119
120    Pose circleStart{-0.15, 0.15, 0.0};
121
122    // Tilts the pose to -30 degrees
123    circleStart.orientation = AngleAxis<double>{
124        rad(-30.0), Vector3d{0, 1, 0}};
125
126    Pose circleMiddle{-0.20, 0.20, 0.0};
127    circleMiddle.orientation = AngleAxis<double>{
128        rad(93.8409657), Vector3d{0.2505628, -0.2505628, 0.9351131}};
129
130    Pose circleEnd{-0.25, 0.15, 0.0};
131    circleEnd.orientation = AngleAxis<double>{
132        rad(+180.0), Vector3d{0.258819, 0, 0.9659258}};
133
134    // Moves the Pose 10 mm in -Z direction
135    Affine3d circleAdd = Affine3d::Identity();
```

```
136     circleAdd.translation() << 0.0, 0.0, 0.01;
137
138     circleStart = Pose{circleAdd.inverse()}.transform(circleStart);
139     circleMiddle = Pose{circleAdd.inverse()}.transform(circleMiddle);
140     circleEnd = Pose{circleAdd.inverse()}.transform(circleEnd);
141
142     // Transforming to the weld part
143     circleStart.transformThis(workObject);
144     circleMiddle.transformThis(workObject);
145     circleEnd.transformThis(workObject);
146
147     var circular1 = make_shared<CircularWay>(
148         circleStart, circleMiddle, circleEnd);
149     circular1->executionConditional = speedZoneConditional;
150     circular1->startTask = turnOn;
151
152     circleMiddle.position << -0.20, 0.10, 0.0;
153     circleMiddle.orientation = AngleAxis<double>{
154         rad(93.8409657), Vector3d{-0.2505628, -0.2505628, -0.9351131}};
155     circleMiddle = Pose{circleAdd.inverse()}.transform(circleMiddle);
156     circleMiddle.transformThis(workObject);
157
158     var circular2 = make_shared<CircularWay>(
159         circleEnd, circleMiddle, circleStart);
160     circular2->executionConditional = speedZoneConditional;
161     circular2->endTask = turnOff;
162
163     // Creates a Path with the two WayPoints
164     Path pathCircle{{circular1, circular2}};
165
166     //
167     // Spot welding
168     //
169
170     Pose spotPoint{-0.202, 0.15, 0};
171     spotPoint.transformThis(workObject);
172
173     var pointWeld = make_shared<LinearWay>(spotPoint, spotPoint);
174     pointWeld->executionConditional = speedZoneConditional;
175     pointWeld->startTask = turnOn;
176     pointWeld->endTask = turnOff;
177     Path pathSpot{{pointWeld}};
178
179     val trajectory = pathPlanning.calculateTrajectory({
180         pathLinear, pathCircle, pathSpot});
181
182     cout << "trajectory planned: " <<
183         boolalpha << trajectory.successfullyPlanned() << endl;
184     if (trajectory) {
185         if(runTime) {
186             // Send the whole trajectory to the robot
187             connection.trajectoryCommandBased(trajectory);
188             connection.start();
189         }
190         cout << "### jointBasedProgram ###\n\n";
191         cout << postProcessor.jointBasedProgram(trajectory) << endl;
192         cout << "### commandBasedProgram ###\n\n";
193         cout << postProcessor.commandBasedProgram(trajectory) << endl;
194         cout << "\n\n";
```

```

196         visualisation.visualizeTrajectory(trajectory);
197     }
198     cout << "\n\n\nProgram finished\n";
199     cout << "Press Enter to close sockets and shutdown program..." << endl;
200     getchar();
201     return 0;
202 }
```

Quellcode 16 zeigt den dazugehörigen generierten RAPID-Programmcode aus der Funktion **commandBasedProgram**.

#### Quellcode 16: Generierter RAPID-Programmcode

```

1 MODULE pathfinder
2 PROC main()
3 MoveAbsJ [[0, -22.198, 38.6944, 0, 51.5034, 0],
4           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
5 WaitTime \InPos,0;
6 ! SetDO do1, 1;
7 MoveL [[750, 300, 710], [0, 0, 1, 0], [0, -1, 0, 0],
8           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v50, z5, tWeldGun;
9 WaitTime \InPos,0;
10 ! SetDO do1, 0;
11 MoveAbsJ [[21.6256, -22.5832, 42.5936, -26.2795, 33.507, -154.402],
12           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
13 MoveAbsJ [[17.4907, -28.8529, 51.4168, -40.2121, 21.5208, -310.836],
14           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
15 WaitTime \InPos,0;
16 ! SetDO do1, 1;
17 MoveC [[900, 205, 718.66], [0.183013, 0.683013, 0.683013, -0.183013],
18           [0, -1, -3, 0], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]],
19           [[955, 150, 718.66], [0, 0.965926, 0, -0.258819],
20           [0, 0, -2, 0], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]],
21           v50, z5, tWeldGun;
22 MoveC [[900, 95, 718.66], [0.183013, -0.683013, 0.683013, 0.183013],
23           [-1, 0, -2, 0], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]],
24           [[845, 150, 718.66], [0.258819, 0, 0.965926, 0],
25           [0, -1, 0, 0], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]],
26           v50, z5, tWeldGun;
27 WaitTime \InPos,0;
28 ! SetDO do1, 0;
29 MoveAbsJ [[12.9619, -29.8148, 48.7921, -40.9601, 23.8237, 54.658],
30           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
31 MoveAbsJ [[16.0465, -22.5702, 37.8838, -34.9512, 19.4408, 67.262],
32           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
33 MoveAbsJ [[14.5123, -18.6739, 37.843, -20.229, 36.7024, 32.4272],
34           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
35 MoveAbsJ [[11.1906, -10.6651, 30.8571, -5.58589, 48.3226, 14.1149],
36           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
37 WaitTime \InPos,0;
38 ! SetDO do1, 1;
39 MoveL [[902, 150, 710], [0, 0, 1, 0], [0, -1, 0, 0],
40           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v50, z5, tWeldGun;
41 WaitTime \InPos,0;
42 ! SetDO do1, 0;
43 MoveAbsJ [[9.73967, -16.4429, 28.6714, -5.40921, 49.4297, 7.44855],
44           [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
45 MoveAbsJ [[0, -70, 40, 0, 60, 0],
```

```

46      [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]], v100, z5, tWeldGun;
47  ENDPROC
48  ENDMODULE

```

Mithilfe von Quellcode 17 kann der Roboter zur Laufzeit Befehle entgegennehmen:

**Quellcode 17:** RAPID-Programmcode zur Ansteuerung des Roboters zur Laufzeit

```

1  MODULE pathfinder

3    RECORD ListItem
4      string procedure;
5      rawbytes messageData;
6    ENDRECORD

8    CONST num LIST_MAX_SIZE := 16384;
9    VAR ListItem list{LIST_MAX_SIZE};
10   VAR num listSize := 0;

12  VAR rawbytes messageData;

14  VAR speeddata speed := v100;
15  VAR zonedata zone := fine;

17  ! Welding tool;
18  PERS tooldata tool := [TRUE, [[49.841566361, 0, 372.194826196],
19                           [0.981627183, 0, 0.190808995, 0]],
20                           [2, [0, 0, 100], [0, 1, 0, 0], 0, 0, 0]];

22  CONST num JOINT_DATA_SIZE := 24;
23  CONST num ROB_DATA_SIZE := 40;
24  CONST num SPEED_DATA_SIZE := 4;
25  CONST num ZONE_DATA_SIZE := 4;

27  FUNC jointtarget jointFromData(VAR rawbytes bytes, \num offset)
28    VAR jointtarget joint;
29    VAR num off := 0;
30    IF Present(offset) off := offset;
31  !
32    joint.extax := [9E9, 9E9, 9E9, 9E9, 9E9, 9E9];
33    UnpackRawBytes bytes, 1+off, joint.robax.rax_1, \Float4;
34    UnpackRawBytes bytes, 5+off, joint.robax.rax_2, \Float4;
35    UnpackRawBytes bytes, 9+off, joint.robax.rax_3, \Float4;
36    UnpackRawBytes bytes, 13+off, joint.robax.rax_4, \Float4;
37    UnpackRawBytes bytes, 17+off, joint.robax.rax_5, \Float4;
38    UnpackRawBytes bytes, 21+off, joint.robax.rax_6, \Float4;
39    RETURN joint;
40  ENDFUNC

41  FUNC robtarget robFromData(VAR rawbytes bytes, \num offset)
42    VAR robtarget rob;
43    VAR num off := 0;
44    IF Present(offset) off := offset;
45  !
46    rob.extax := [9E9, 9E9, 9E9, 9E9, 9E9, 9E9];
47    rob.robconf.cfx := 0;
48    UnpackRawBytes bytes, 1+off, rob.trans.x, \Float4;
49    UnpackRawBytes bytes, 5+off, rob.trans.y, \Float4;
50    UnpackRawBytes bytes, 9+off, rob.trans.z, \Float4;
51    UnpackRawBytes bytes, 13+off, rob.rot.q1, \Float4;
52    UnpackRawBytes bytes, 17+off, rob.rot.q2, \Float4;
53    UnpackRawBytes bytes, 21+off, rob.rot.q3, \Float4;

```

```
53     UnpackRawBytes bytes, 25+off, rob.rot.q4, \Float4;
54     UnpackRawBytes bytes, 29+off, rob.robconf.cf1, \Float4;
55     UnpackRawBytes bytes, 33+off, rob.robconf.cf4, \Float4;
56     UnpackRawBytes bytes, 37+off, rob.robconf.cf6, \Float4;
57     RETURN rob;
58 ENDFUNC

59 FUNC speeddata speedFromData(VAR rawbytes bytes, \num offset)
60     VAR speeddata speed := v100;
61     VAR num off := 0;
62     IF Present(offset) off := offset;

63     UnpackRawBytes bytes, 1+off, speed.v_tcp, \Float4;
64     RETURN speed;
65 ENDFUNC

66 FUNC zonedata zoneFromData(VAR rawbytes bytes, \num offset)
67     VAR zonedata zone := z5;
68     VAR num zoneSize;
69     VAR num off := 0;
70     IF Present(offset) off := offset;

71     UnpackRawBytes bytes, 1+off, zoneSize, \Float4;

72     IF zoneSize <= 0.0      RETURN fine;
73     IF zoneSize < 1.0       RETURN z0;
74     IF zoneSize < 5.0       RETURN z1;
75     IF zoneSize < 10.0      RETURN z5;
76     IF zoneSize < 20.0      RETURN z10;
77     IF zoneSize < 30.0      RETURN z20;
78     IF zoneSize < 40.0      RETURN z30;
79     IF zoneSize < 50.0      RETURN z40;
80     IF zoneSize < 60.0      RETURN z50;
81     IF zoneSize < 80.0      RETURN z60;
82     IF zoneSize < 100.0     RETURN z80;
83     IF zoneSize < 150.0     RETURN z100;
84     IF zoneSize < 200.0     RETURN z150;
85     RETURN z200;
86 ENDFUNC

87 PROC SetSpeed0()
88     speed := speedFromData(messageData);
89 ENDPROC

90 PROC SetZone0()
91     zone := zoneFromData(messageData);
92 ENDPROC

93 PROC MoveAbsJ0()
94     MoveAbsJ jointFromData(messageData), speed, zone, tool;
95 ENDPROC

96 PROC MoveL0()
97     MoveL robFromData(messageData), speed, zone, tool;
98 ENDPROC

99 PROC MoveC0()
100    MoveC robFromData(messageData),
```

```
112                 robFromData(messageData, \offset:=ROB_DATA_SIZE),
113                 speed, zone, tool;
114             ENDPROC

117     PROC TurnOnWeldingGun0()
118         WaitTime \InPos, 1;
119         ! SetDO doi 1;
120     ENDPROC

122     PROC TurnOffWeldingGun0()
123         WaitTime \InPos, 0;
124         ! SetDO doi 1;
125     ENDPROC

128     PROC AddToList0()
129         VAR num procedureLength;

131         listSize := listSize+1;
132         UnpackRawBytes messageData, 1,
133             procedureLength, \IntX:=USINT;
134         UnpackRawBytes messageData, 2,
135             list{listSize}.procedure, \ASCII:=procedureLength;
136         ClearRawBytes list{listSize}.messageData;
137         CopyRawBytes messageData,
138             2+procedureLength, list{listSize}.messageData, 1;
139     ENDPROC

141     PROC ClearList0()
142         listSize := 0;
143     ENDPROC

145     PROC StartList0()
146         FOR i FROM 1 TO listSize DO
147             CopyRawBytes list{i}.messageData, 1, messageData, 1;
148             CallByVar list{i}.procedure, 0;
149         ENDFOR

151         ClearList0;
152     ENDPROC

155     PROC main()
156         VAR socketdev serverSocket;
157         VAR socketdev clientSocket;

159         CONST string ip := "192.168.135.69";
160         CONST num port := 12345;

162         VAR string procedure;

164         VAR num procedureLength;
165         VAR num dataLength;

167         SocketCreate serverSocket;
168         SocketBind serverSocket, ip, port;
169         SocketListen serverSocket;
```

```

171     SocketAccept serverSocket, clientSocket, \Time:=WAIT_MAX;

173     WHILE TRUE DO
174         ClearRawBytes messageData;
175         SocketReceive clientSocket, \RawData:=messageData,
176             \ReadNoOfBytes:=3, \Time:=WAIT_MAX;

178         UnpackRawBytes messageData, 1, procedureLength, \IntX:=USINT;
179         UnpackRawBytes messageData, 2, dataLength, \IntX:=UINT;
180         ClearRawBytes messageData;
181         SocketReceive clientSocket, \RawData:=messageData,
182             \ReadNoOfBytes:=procedureLength, \Time:=WAIT_MAX;

184         UnpackRawBytes messageData, 1, procedure, \ASCII:=procedureLength;
185         ClearRawBytes messageData;
186         IF dataLength > 0
187             SocketReceive clientSocket, \RawData:=messageData,
188                 \ReadNoOfBytes:=dataLength, \Time:=WAIT_MAX;

190         CallByVar procedure, 0;
191     ENDWHILE
192 ENDPROC
193 ENDMODULE

```

## D Testprogramme der Eingabeschnittstelle

Das folgende Programm zeigt die Programmierung des Schweißbauteils, mithilfe der Eingabeschnittstelle.

**Quellcode 18:** C++-Programm des Schweißbauteils mit der Eingabeschnittstelle

```

1 #include <iostream>
2 #include <chrono>

4 #include "inputinterface.h"
5 #include "rapidabb/rapidinputinterface.h"

7 #define var auto
8 #define val const auto
9 #define ref auto&&
10 #define viu const auto&

12 using namespace pf;
13 using namespace pf::ii;
14 using namespace pf::rapid;
15 using namespace std;
16 using namespace std::chrono;
17 using namespace Eigen;

19 constexpr val runTime = false;

21 int main() {
22     if(runTime)
23         setup(new IRB2600ID{"192.168.135.69", 12345});
24     else
25         setup(new IRB2600ID{});

```

```
27     useMilliMeters();
28     useDegrees();

30     speed(50);
31     zone(5);

33     // Weldgun on / off tasks      "RunTime procedure", "PostProcessor code"
34     var turnOn = make_shared<RobotTask>("TurnOnWeldingGun",
35         "WaitTime \\InPos,0;\n! SetDO do1, 1;");
36     var turnOff = make_shared<RobotTask>("TurnOffWeldingGun",
37         "WaitTime \\InPos,0;\n! SetDO do1, 0;");

39     // Position      Orientation as Quaternion
40     // x, y, z,      w, i, j, k
41     setCoordinationSystem(Pose{700, 0, 710, 0, 0, 1, 0});

43     //
44     // V-seam welding
45     //
46     start(x(-50));
47     task(turnOn);
48     moveL(y(300));
49     task(turnOff);

51     //
52     // Round fillet welding (needs 2 CircularWays,
53     // because a CircularWay must be < 360 degrees
54     //
55     nextPath();
56     start(x(-145), y(150), z(-8.66025), euler(0, -30, 0));

58     task(turnOn);
59     moveCvia(addX(-55), addY(55), addEuler(0, 0, 90));
60     moveCto(addX(-55), addY(-55), addEuler(0, 0, 90));

62     moveCvia(addX(55), addY(-55), addEuler(0, 0, 90));
63     moveCto(addX(55), addY(55), addEuler(0, 0, 90));
64     task(turnOff);

66     //
67     // Spot welding
68     //
69     nextPath();
70     resetPose();
71     start(x(-202), y(150));
72     task(turnOn);
73     moveL(x(-198));
74     task(turnOff);

76     Trajectory trajectory = getPathPlanning().calculateTrajectory(getPaths());
77     if (trajectory) {
78         if(runTime) {
79             // Send the whole trajectory to the robot
80             getRunTimeConnection().trajectoryCommandBased(trajectory);
81             getRunTimeConnection().start();
82         }
83         getVisualisation().visualizeTrajectory(trajectory);
84     }
```

```

85     cout << "\n\n\nProgram finished\n";
86     cout << "Press Enter to close sockets and shutdown program..." << endl;
87     getchar();
88     return 0;
89 }

```

Mithilfe der Python-Anbindung sieht das Python-Programm wie folgt aus:

**Quellcode 19:** Python-Programm des Schweißbauteils

```

1  from pypf import *
2
3  RUNTIME = True
4
5  if RUNTIME:
6      setupIRB2600ID("192.168.135.69", 12345)
7  else:
8      setupIRB2600ID()
9
10 useMilliMeters()
11 useDegrees()
12
13 speed(50)
14 zone(5)
15
16 # Weldgun on / off tasks      "RunTime procedure", "PostProcessor code"
17 turnOn = createTask("TurnOnWeldingGun", "WaitTime \\InPos,0;\n! SetDO do1, 1;")
18 turnOff = createTask("TurnOffWeldingGun", "WaitTime \\InPos,0;\n! SetDO do1, 0;")
19
20 # Position      Orientation as Quaternion
21 # x, y, z,      w, i, j, k
22 setCoordinationSystem(700, 0, 710, 0, 0, 1, 0)
23
24 #
25 # V-seam welding
26 #
27
28 start(x(-50))
29 task(turnOn)
30 moveL(y(300))
31 task(turnOff)
32
33 #
34 # Round fillet welding (needs 2 CircularWays,
35 # because a CircularWay must be < 360 degrees
36 #
37
38 nextPath()
39 start(x(-145), y(150), z(-8.66025), euler(0, -30, 0))
40
41 task(turnOn)
42 moveCvia(addX(-55), addY(55), addEuler(0, 0, 90))
43 moveCto(addX(-55), addY(-55), addEuler(0, 0, 90))
44
45 moveCvia(addX(55), addY(-55), addEuler(0, 0, 90))
46 moveCto(addX(55), addY(55), addEuler(0, 0, 90))
47 task(turnOff)
48
49 #
50 # Spot welding

```

```
51  #
53  nextPath()
54  resetPose()
55  start(x(-202), y(150))
56  task(turnOn)
57  moveL(x(-198))
58  task(turnOff)

61  if not calculate():
62      print("Trajectory could not be planned")
63  else:

65      if RUNTIME:
66          print("Sending commands...")
67          sendCommandBased()

69      print("### jointBasedProgram ##\n\n")
70      print(jointBasedProgram())

72      print("### commandBasedProgram ##\n\n")
73      print(commandBasedProgram())

75      visualize()

78  print("\n\n\nProgram finished\nPress Enter to close")
79  raw_input("")
```

## E Dokumentation der entwickelten Roboterbibliothek

Auf der beigefügten CD befindet sich neben dem Quellcode und einem Simulationsvideo, die Dokumentation der Roboterbibliothek. Diese wurde mit dem Programm Doxygen [21] erstellt und ist als Website in HTML oder als PDF verfügbar.