Lehrstuhl für Raumfahrttechnik
Prof. Prof. h.c. Dr. Dr. h.c.
Ulrich Walter

Technische Universität München

# Semesterarbeit

# Mathematische Beschreibung der Kinematik des RACOON-Lab und Implementierung einer Inversen Kinematik als Softwaremodul

## RT-SA 2017/11

Autor:

Aschenbrenner, Fabian


Betreuer: Dipl.-Ing. Martin Dziura
Lehrstuhl für Raumfahrttechnik / Institute of Astronautics
Technische Universität München

# Danksagung

Ich möchte mich beim Lehrstuhl für Angewandte Mechanik und insbesondere bei Felix Sygulla und Daniel Wahrmann bedanken, die mir jeder Zeit geholfen haben, wenn Fragen zum Thema inverse Kinematik aufgekommen sind. Außerdem haben sie mir ein MATLAB-Skript zur Verfügung gestellt, das für die Lehrveranstaltung Roboterdynamik gedacht war. Ihre Hilfe hat einen großen Beitrag zum Gelingen dieser Arbeit beigetragen.

# Zusammenfassung

Für das RACOON-Lab, das eine Simulationsumgebung für On-Orbit-Servicing ist, soll die inverse Kinematik neu konzeptioniert werden. Bei der inversen Kinematik werden aus den Eingangsdaten (Endposition und Ausrichtung) Gelenkwinkel bzw. -koordinaten berechnet, sodass der Roboter die relative Lage zweier Satelliten zueinander in Echtzeit abbildet. Das RACOON-Lab besitzt mehr Freiheitsgrade als Größen, die zur eindeutigen Bestimmung der relativen Lage nötig wären, was Redundanz zur Folge hat. Das ermöglicht es dem System wiederum sich freier im Raum zu bewegen.

Die Abstrahierung und Beschreibung des Systems ist an die DH-Parametrisierung angelehnt, bei der die Transformationsmatrix zweier Koordinatensysteme nur aus Rotationen um die x- bzw. z-Achse und Verschiebungen in diese Achsen gebildet wird. Aus der kinematischen Kette, d.h. der Reihenfolge der Gelenke, entsteht ein Set an Parametern, die die Kinematik des Systems eindeutig beschreiben. Daraus erhält man die Beziehung zwischen den Freiheitsgraden des Roboters und der absoluten Position bzw. Ausrichtung der beiden Satelliten im Raum. Mit Hilfe dieser Beschreibung und den Eingangsdaten werden dann die optimalen Gelenkwinkel berechnet. Die physischen Grenzen des Systems werden zusätzlich durch eine Gewichtungsmatrix mitberücksichtigt, indem dem Roboter der Freiheitsgrad, der sich an seiner Grenze befindet, dann nicht mehr zur Verfügung steht.

Die Implementierung erfolgt aufgrund der einfachen Visualisierung der Ergebnisse in MATLAB und da der Lehrstuhl für Angewandte Mechanik ein rohes Skript für einen nicht-redundanten Roboter mit drei rotatorischen Freiheitsgraden zur Verfügung gestellt hat, welches für das RACOON-Lab angepasst wird.

Neben systematischen Problemen stellen sich Schwierigkeiten bei der Kompilierung und Einbindung des Programms in Visual Studio dar. Die Ergebnisse in MATLAB zeigen eine erfolgreiche Beschreibung der Kinematik, jedoch funktioniert die Kommunikation zwischen dem MATLAB Programm und Visual Studio noch nicht optimal, sodass der Algorithmus für die Simulation im RACOON-Lab sein volles Potential nicht entfalten kann.

# Abstract

The RACOON-Lab is a simulation environment for On-Orbit-Servicing, which can represent the relative position and orientation of two satellites in real-time. This thesis deals with a new description of the inverse kinematics of the system in order to calculate the necessary values for the links of the robot with the input data from an orbit simulation. The robot has more degrees of freedom than required parameters to describe the relative position and orientation, which causes redundancy.

The abstraction and description of the system is close to DH-parametrisation. The transformation matrices are built with rotation and translation around the x- and z-axis. With the kinematic chain, that means the order of the degrees of freedom, and the DH-parametrisation you get a set of parameters, which characterize explicit the complete kinematic of the system. The result is the connection between the links and the absolute parameters in the reference frame to describe the relative position and orientation of two satellites. The physical boundaries are also included with a weighting factor, that withdraws the degree of freedom, which is close to its boundary. With the input data and this connection, the algorithm can calculate the optimal values for the links of the robot.

The implementation is done in MATLAB, due to an easy possibility to show the results in graphs. Moreover, the chair of applied mechanics provides a raw code, which solves a non-redundant robot with three rotational degrees of freedom. This code is adapted and extended for the problem of this thesis.

Beside systematic issues, there are difficulties in compiling the code in C# and including it in Visual Studio, where the simulation is done. The results in MATLAB show a successful description of the inverse kinematic, though the communication between the algorithm and Visual Studio does not work perfect yet. In order to completely meet the requirements, the last step is to optimize the integration of the code in future theses.

# Inhaltsangabe

**1 EINFÜHRUNG 1**

**2 PROBLEMSTELLUNG UND ZIEL DER ARBEIT 2**

**3 SYSTEMBESCHREIBUNG 3**

**3.1 Hardware 3**

**3.2 Aktueller Stand des Axismappers 5**

**4 EINFÜHRUNG IN DIE ROBOTIK 6**

**4.1 Darstellungsweisen und Transformationen 6**
4.1.1 Position, Orientierung und Koordinatensysteme 6
4.1.2 Räumliche Transformationen 7
4.1.3 Denavit-Hartenberg-Konvention 9
4.1.4 Denavit-Hartenberg-Transformation 10

**4.2 Direkte Kinematik 12**

**4.3 Inverse Kinematik 12**

**4.4 Planung der Trajektorien 13**

**5 KINEMATISCHE BESCHREIBUNG DES RACOON-LAB 15**

**5.1 Systemmodellierung 15**

**5.2 Problemformulierung 22**

**5.3 Lösung des Problems 23**

**6 IMPLEMENTIERUNG 32**

**6.1 Implementierung in MATLAB 32**

**6.2 Kompilierung und Einbindung in Visual Studio 34**

**7 VERIFIKATION 37**

**7.1 Verifikation in MATLAB 37**
7.1.1 Scenario 1 37
7.1.2 Scenario 2 41
7.1.3 Scenario 3 43

**7.2 Verifikation im RACOON-Lab 46**

**8 ZUSAMMENFASSUNG 49**

**9 LITERATUR 51**

# Abbildungsverzeichnis

Abb. 1–1: Architektur des RACOON-Lab (Lehrstuhl für Raumfahrttechnik: RacoonLab) ..... 1
Abb. 3–1: Foto des RACOON-Lab mit allen Freiheitsgraden ..... 3
Abb. 3–2: Kopf des Chasers ohne Verbindungsstange zur z-Achse ..... 4
Abb. 3–3: Komplettes Target ..... 5
Abb. 3–4: Vergrößerung des Targets ..... 5
Abb. 4–1: Position und Orientierung eines Punktes (angelehnt an Abb. in (Craig 2014: 21)) ..... 7
Abb. 4–2: Die Denavit-Hartenberg-Parameter (Abbildung nach (Craig 2014)) ..... 9
Abb. 4–3: Mögliche Pfade für einen Winkel θ (Craig 2014: 204) ..... 13
Abb. 5–1: Prinzipskizze des RACOON-Labs mit den Freiheitsgraden und dem rot angedeuteten „C“ ..... 20
Abb. 5–2: Baumstruktur der Freiheitsgrade des RACOON-Lab (Links: Chaser; Rechts: Target) ..... 20
Abb. 5–3: Flowdiagramm für den Maincode ..... 24
Abb. 5–4: Drehung des Ortsvektors R auf die x-Achse ..... 25
Abb. 5–5: Funktionsweise des Zählers für die Erweiterung des Winkelbereichs ..... 26
Abb. 5–6: Beispielhafte Gewichtungsfunktion für die Grenzen -0,5 und 2,0 ..... 27
Abb. 5–7: Veranschaulichung der "Singularity Avoidance" ..... 28
Abb. 5–8: 90° Rotation des Targets um die y-Achse mit Sinularity Avoidance ..... 29
Abb. 5–9: 90° Rotation des Targets um die y-Achse ohne Sinularity Avoidance ..... 29
Abb. 5–10: Algorithmus zur Berechnung der Inversen Kinematik ..... 31
Abb. 7–1: Elliptische Sollbahn des Chasers ..... 38
Abb. 7–2: Sollwerte für die Rotation des Targets um die z-Achse ..... 38
Abb. 7–3: Gelenkkoordinaten q für den Chaser mit dt=1ms ..... 38
Abb. 7–4: Arbeitsraumkoordinaten w für den Chaser mit dt=1ms ..... 38
Abb. 7–5: Gelenkkoordinaten q für das Target mit dt=1ms ..... 39
Abb. 7–6: Arbeitsraumkoordinaten w für das Target mit dt=1ms ..... 39
Abb. 7–7: Gelenkkoordinaten q für den Chaser mit dt=2.5ms ..... 39
Abb. 7–8: Arbeitsraumkoordinaten w für den Chaser mit dt=2.5ms ..... 39
Abb. 7–9: Gelenkkoordinaten q für das Target mit dt=2.5ms ..... 39
Abb. 7–10: Arbeitsraumkoordinaten w für das Target mit dt=2.5ms ..... 39
Abb. 7–11: Gelenkkoordinaten q für den Chaser mit dt=5ms ..... 40
Abb. 7–12: Arbeitsraumkoordinaten w für den Chaser mit dt=1ms ..... 40
Abb. 7–13: Gelenkkoordinaten q für das Target mit dt=5ms ..... 40
Abb. 7–14: Arbeitsraumkoordinaten w für das Target mit dt=5ms ..... 40
Abb. 7–15: Sollbahn des Chasers Scenario 2 ..... 42
Abb. 7–16: Sollwinkel des Targets Scenario 2 ..... 42

Abb. 7–17: Gelenkkoordinaten q für den Chaser Scenario 2 ..... 43
Abb. 7–18: Arbeitsraumkoordinaten w für den Chaser Scenario 2 ..... 43
Abb. 7–19: Gelenkkoordinaten q für das Target Scenario 2 ..... 43
Abb. 7–20: Arbeitsraumkoordinaten w für das Target Scenario 2 ..... 43
Abb. 7–21: Sollbahn des Chasers Scenario 3 ..... 44
Abb. 7–22: Sollwinkel des Targets Scenario 3 ..... 44
Abb. 7–21: Gelenkkoordinaten q für den Chaser Scenario 3 ..... 45
Abb. 7–22: Arbeitsraumkoordinaten w für den Chaser Scenario 3 ..... 45
Abb. 7–23: Gelenkkoordinaten q für das Target Scenario 3 ..... 45
Abb. 7–24: Arbeitsraumkoordinaten w für das Target Scenario 3 ..... 45

# Tabellenverzeichnis

Tab. 5–1: DH-Parameter für den Chaser ..... 16
Tab. 5–2: DH-Parameter für das Target ..... 17
Tab. 5–3: Koordinatensysteme für den Chaser nach Transformation mit DH ..... 18
Tab. 5–4: Koordinatensysteme für das Target nach Transformation mit DH ..... 19
Tab. 5–5: Achsenbezeichnungen ..... 21
Tab. 5–6: Grenzen der Freiheitsgrade der Anlage ..... 22
Tab. 6–1: Auflistung der Bestandteile der Klasse Roboter ..... 33
Tab. 6–2: Verknüpfung der Spaltenindizierung mit den Bezeichnungen ..... 35
Tab. 7–1 Randbedingungen für drei Berechnungen in MATLAB ..... 37
Tab. 7–2: Ausgewertete Daten für die Berechnungen mit dt=1ms/2.5ms/5ms ..... 40
Tab. 7–3: Randbedingungen für Scenario 2 ..... 41
Tab. 7–4: Allgemeine Daten zu Scenario 3 ..... 44
Tab. 7–5: Ergebnisse und Startwerte für Scenario 3 ..... 45
Tab. 7–6: Eingangswerte für die Anlage ..... 46
Tab. 7–7: Position des Roboters aus der Berechnung des Algorithmus' ..... 47
Tab. 7–8: Lösung für die Gelenkkoordinaten ..... 47

# Symbole und Formelzeichen

| | | |
|---|---|---|
| $q$ | [m]/[-] | Freiheitsgrad |
| $w$ | [m]/[-] | Arbeitsraumkoordinate |
| $\dot{q}$ | [1/s] | Gelenkgeschwindigkeit |
| $\dot{w}$ | [1/s]/[m/s] | Arbeitsraumgeschwindigkeit |
| $w_d$ | [m]/[-] | Zielarbeitsraumkoordinaten |
| $\dot{w}_d$ | [m/s]/[1/s] | Geschwindigkeit der Zielarbeitsraumkoordinaten |
| $A$ | [-] | Transformationsmatrix |
| $D_i$ | [-] | DH-Transformation |
| $B_0$ | [-] | Inertialsystem |
| $B_i$ | [-] | Relativsystem |
| $J_w$ | [-] | Jakobimatrix |
| $J_R$ | [-] | rot. Jakobimatrix |
| $J_T$ | [-] | transl. Jakobimatrix |
| $W$ | [-] | Gewichtungsmatrix |
| $K$ | [-] | positiv definite Matrix |
| $x$ | [-] | Variable |
| $r_{vi}$ | [-] | relativer Verschiebungsvektor |
| $\tilde{r}_{vi}$ | [-] | Tilde-Matrix aus $r_{vi}$ |
| $e_{x,y,z}$ | [-] | Einheitsvektoren |

| | | |
|---|---|---|
| $\alpha$ | [°] | Winkel |
| $\beta$ | [°] | Winkel |
| $\gamma$ | [°] | Winkel |
| $\theta$ | [°] | Winkel |
| $a_{i-1,i}$ | [-] | DH-Parameter |
| $\theta_{i-1,i}$ | [°] | DH-Parameter |
| $\alpha_{i-1,i}$ | [°] | DH-Parameter |
| $d_{i-1,i}$ | [-] | DH-Parameter |
| $a_i$ | [-] | Koeffizient |
| $g$ | [-] | Grenze für Gelenk |
| $h$ | [-] | Grenze für Gelenk |
| ${}_ir_P$ | [-] | Ortsvektor |
| $p_i$ | [m]/[-] | Ortskoordinate |
| $t$ | [s] | Zeit |
| $dt$ | [s] | Zeitintervall |
| $T$ | [s] | Gesamtzeit |
| $f$ | [-] | Laufparameter |
| $\omega$ | [1/s] | Winkelgeschwindigkeit |
| $v$ | [m/s] | transl. Geschwindigkeit |

# Abkürzungen

| | | | |
|---|---|---|---|
| Abb. | Abbildung | Tab. | Tabelle |
| ECI | Earth Center Inertial | DH | Denavit-Hartenberg |
| KOS | Koordinatensystem | AMM | Lehrstuhl für Angewandte Mechanik |
| transl. | translatorisch | | |
| rot. | rotatorisch | | |

# 1 Einführung

Der Lehrstuhl für Raumfahrttechnik besitzt eine Hardware, die eine Simulationsumgebung für On-Orbit-Servicing darstellt, das RACOON (Real-time Attitude Control and OnOrbit Navigation) Labor. Mit Hilfe 12 kinematischer Achsen kann das System den Anflug zweier Satelliten im Orbit abbilden. Die Berechnungen dafür stammen aus einer dynamischen Orbitsimulation, welche Position und Lage beider Satelliten in Echtzeit und in ECI-Koordinaten zur Verfügung stellt. Mögliche Missionen reichen von Wartungsarbeiten bis zur Beseitigung von Weltraumschrott.

Um eine realistische Simulationsumgebung gewährleisten zu können, besteht das RACOON-Lab aus einem Mission Control Center, einer Umgebung die Hardware-in-the-Loop Verfahren und reine Software Simulationen ermöglichen. (A. Fleischner, M. Wilde and U. Walter 2012)

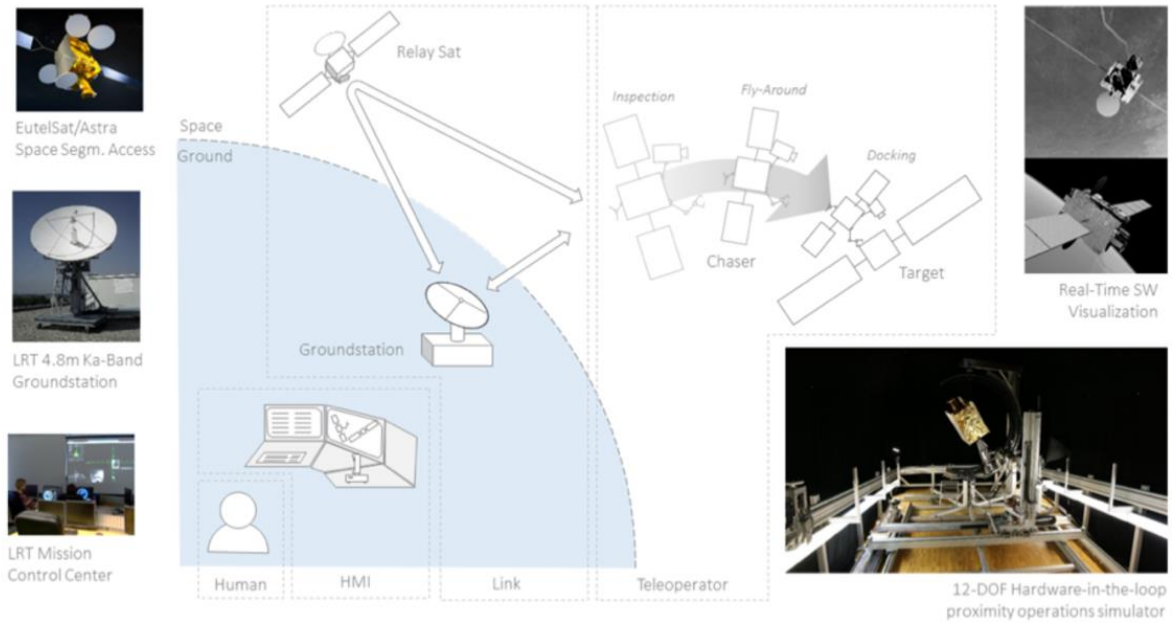



Abb. 1–1: Architektur des RACOON-Lab (Lehrstuhl für Raumfahrttechnik: RacoonLab)

Das RACOON-Lab basiert auf der Architektur einer teleoperierten Weltraummission, bei der ein Operator über eine Funkverbindung ein robotisches System mit Aktuatoren und Sensoren im Orbit steuert (vgl. Abb. 1–1). Das ermöglicht es orbitale Strukturen aufzubauen oder beschädigte Satelliten zu reparieren. Die Ende-zu-Ende Simulation ermöglicht es das gesamte System vom Roboter im Weltraum bis zum Operator am Boden zu simulieren. Die zu simulierenden Operationen beschränken sich dabei auf den nahen Anflugbereich von 20m ohne Docking oder Einfangen. (Lehrstuhl für Raumfahrttechnik: RacoonLab)

# 2 Problemstellung und Ziel der Arbeit

Für die Simulierung eines Anflugmanövers müssen dem RACOON-Lab Daten zur Verfügung gestellt werden. Diese Informationen umfassen die Ortsvektoren, Geschwindigkeiten und Ausrichtung der beiden Satelliten in ECI-Koordinaten (Earth-centered-inertial). Die Vektoren in ECI-Koordinaten werden zunächst in für die Anlage verwendbare Koordinaten transformiert. Aus diesen Informationen berechnet der Axismapper die benötigten Translationen und Rotationen für die einzelnen Freiheitsgrade, damit die Bewegung der beiden Satelliten detailliert abgebildet werden kann. Das bedeutet, dass der Axismapper über geeignete mathematische Verfahren aus den Anfangs- und Endpositionen der Satelliten die Gelenkwinkel für den Roboter ermittelt. Für die Simulierung der Position und Lage muss die Kinematik des Systems beschrieben werden. Darunter fällt die Bestimmung der Freiheitsgrade und der physikalischen Grenzen der Anlage, der kinematischen Kette und ob das System möglicherweise redundant ist, was für das RACOON-Lab zutrifft. Eine kinematische Kette ist ein System aus starren Körpern, welche durch Gelenke verbunden sind (Siciliano et al. 2009: 39). Ein anschauliches Beispiel stellt der menschliche Arm dar, angefangen vom Schultergelenk bis zu den Fingern. Eine Bewegung des Unterarms hat Einfluss auf alle nachfolgenden Glieder (Hand und Finger), jedoch nicht auf das vorangestellte Glied (Oberarm). Ein Roboter ist kinematisch redundant, wenn für die Erfüllung einer Aufgabe die Anzahl der Freiheitsgrade größer als die Anzahl an benötigten Variablen ist (Siciliano et al. 2009: 87–8).

Das Ziel dieser Arbeit besteht darin, den Axismapper unter der Verwendung von inverser Kinematik neu zu konzeptionieren. Die Kinematik des Systems soll eindeutig beschrieben werden, sodass die Software mit gegebenen Anfangsbedingungen und unter Berücksichtigung von Randbedingungen die optimalen Gelenkwinkel aus der Schar an möglichen Lösungen findet. Der Einfluss jedes Freiheitsgrades soll zudem mit einer Gewichtungsmatrix gesteuert werden können.

Die anfängliche Implementierung und Validierung findet aufgrund der einfachen Überprüfbarkeit und Visulisierung der Ergebnisse in MATLAB statt. Anschließend wird die Berechnung in Visual Studio bzw. das RACOON-Lab implementiert und getestet. Die für die Beschreibung der inversen Kinematik benötigte Mathematik wird in Kapitel 4 vorgestellt und im nachfolgenden Kapitel 5 auf das RACOON-Lab angewendet.

Diese Arbeit verwendet grundlegend die Literatur (Craig 2014) und (Siciliano et al. 2009), da die Vorlesung Roboterdynamik (AMM) auf diesen basiert und mir vom Lehrstuhl für Angewandte Mechanik als Primärliteratur empfohlen wurden.

# 3 Systembeschreibung

Um die Kinematik des Systems richtig erfassen und diese in das bestehende Softwaremodul implementieren zu können, ist eine anfängliche Beschreibung des aktuellen Stands der Hard- und Software nötig. Deshalb wird im Folgenden näher auf den Aufbau der Anlage, den Randbedingungen, der kinematischen Kette sowie den Aufbau des Axismappers eingegangen und wo diese Arbeit ins gesamte Konzept des RACOON-Labs anzusiedeln ist.

## 3.1 Hardware

Die Hardware des RACOON-Labs besteht aus einem Zielsatelliten (Target) und einem zweiten Satelliten (Chaser), der On-Orbit-Servicing am Zielsatelliten verrichten oder diesen aus dem Weltraum entfernen soll. Das Target bleibt dabei fest im Raum stehen und wird nur um fünf Achsen rotiert. Der Chaser hingegen kann sich sowohl translatorisch in drei Richtungen sowie rotatorisch um drei Achsen bewegen. Das bedeutet, dass der Roboter aus zwei Armen besteht und somit keine Kette, sondern eine Baumstruktur aufweist, da die Bewegungen der beiden Satelliten unabhängig voneinander sind. Abb. 3–1 zeigt alle Freiheitsgrade des Roboters ohne systematischer Ordnung, die anschließend erfolgt, auf.

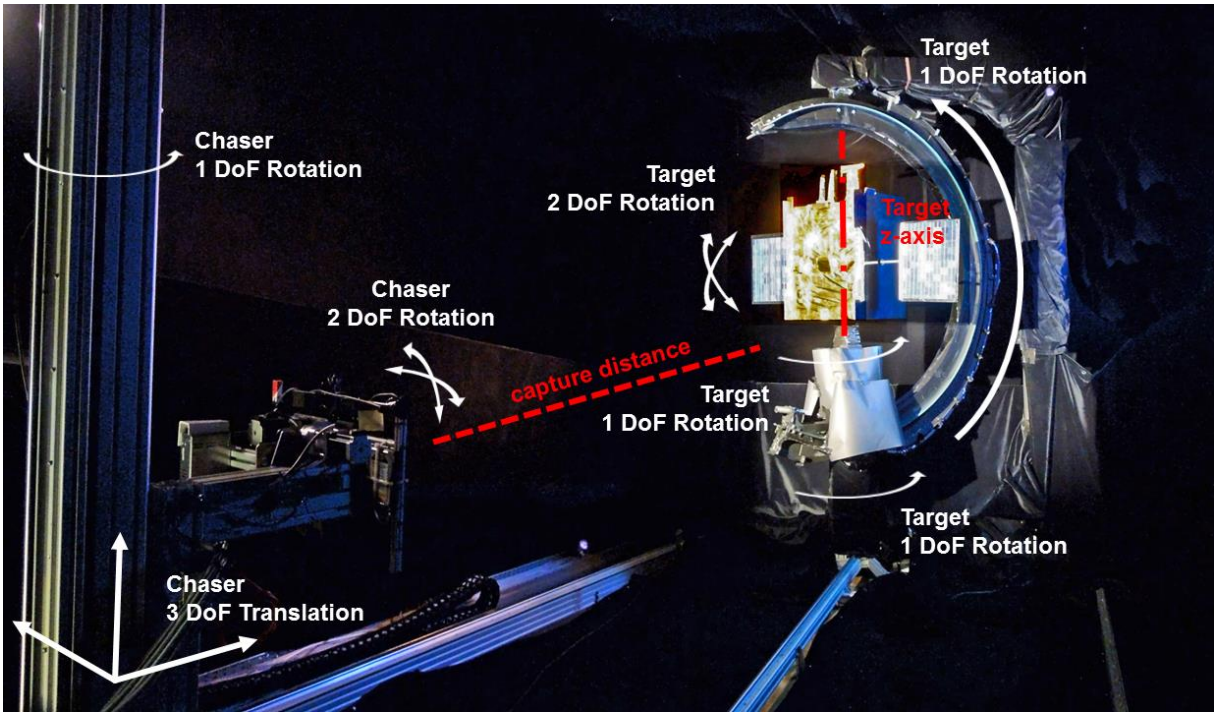



Abb. 3–1: Foto des RACOON-Lab mit allen Freiheitsgraden

Um die kinematische Kette bzw. Baumstruktur zu identifizieren, kann man sich beispielsweise das CATIA-Modell zur Hilfe holen. Für den Chaser kann man die Translation unabhängig von den Rotationen betrachten, da diese bei der Berechnung der Transformationsmatrizen superponieren, was im Kapitel 4 näher beschrieben wird. Im Vorhinein ist zu sagen, dass die xy-Ebene, die „Tischeebene“ und die z-Achse, die Hochachse des RACOON-Labs sind. Für die Translation in x-Richtung bewegt sich

der Chaser zum Target hin bzw. von ihm weg. Die Translation in y-Richtung stellt eine Seitwärtsbewegung dar.

Die Kette für den Chaser beginnt mit der Rotation um die globale z-Achse. Der Kopf des Chasers ist um ca. 0.5 m von der Rotationsachse entfernt und hat dort im Ruhezustand einen Freiheitsgrad um die y-Achse. Bei Verdrehung der z-Achse um 90° wäre dieser ein Freiheitsgrad um die x-Achse, was aber durch die kinematische Kette und den resultierenden Transformationsmatrizen berücksichtigt wird. Den letzten Freiheitsgrad bildet die Rotation des Kopfes um die x-Achse. Abb. 3–2 zeigt die Freiheitsgrade aus einem Ausschnitt des CATIA-Modells. Die Verbindungsstange zur z-Achse ist dort nicht zu sehen, jedoch die Halterung, die sich oberhalb der rot gekennzeichneten y-Achse befindet. Aus dieser Abbildung ist auch klar die ZYX-Reihenfolge der Rotationen zu erkennen.

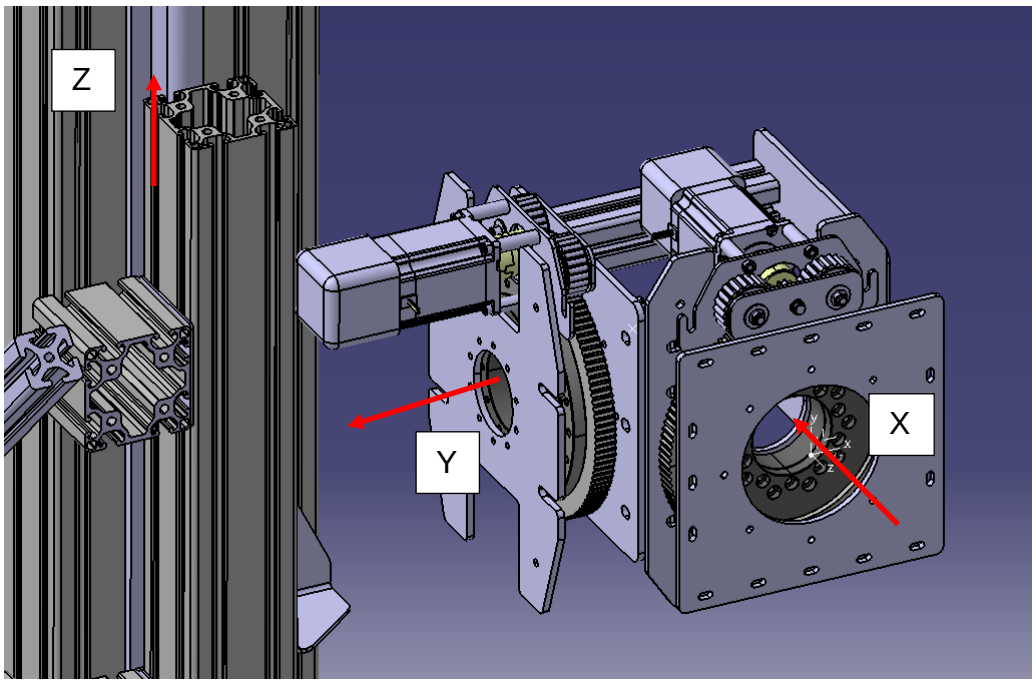



Abb. 3–2: Kopf des Chasers ohne Verbindungsstange zur z-Achse

Das Target besteht aus einem „C“-Segment, welches um die z-Achse rotieren kann. Innerhalb des C’s rotiert der Satellit um die y-Achse (vgl. Abb. 3–3). Für die restlichen Freiheitsgrade wird der Satellit vergrößert dargestellt, damit die Abhängigkeiten klar erkennbar sind. Das CATIA-Modell in Abb. 3–4 zeigt die XYZ-Reihenfolge der Rotationen, nach der Befestigung des Satelliten auf der Platte oberhalb der z-Achse. Ob der Satellit um die jeweilige Achse komplett rotieren kann, lässt sich ebenfalls aus der Abb. 3–2 und Abb. 3–3 ersehen. Die Rotation des C’s um die z-Achse ist uneingeschränkt, wohingegen die Rotation innerhalb des C’s um die y-Achse eingeschränkt ist. Die Rotation des C’s sowie des Satelliten um die x-Achse sind uneingeschränkt, wohingegen die Rotation innerhalb des C’s um die y-Achse eingeschränkt und die Rotationen des Satelliten um die y- bzw. x-Achse stark eingeschränkt sind, wenn man bedenkt, dass die Mechanik im Inneren des Satellitenmodells untergebracht wird (vgl. Abb. 3–4).

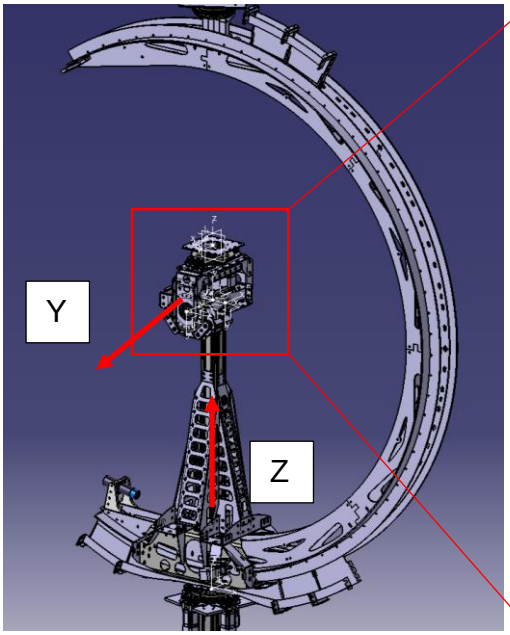



Abb. 3–3: Komplettes Target

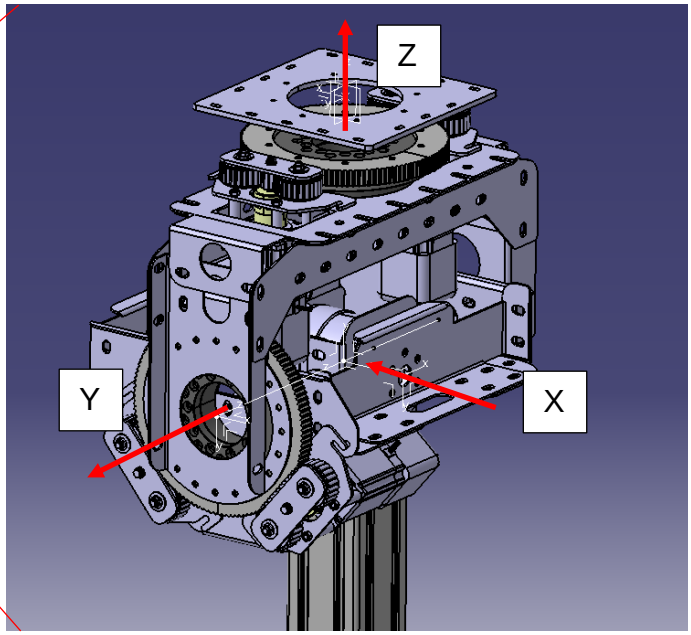



Abb. 3–4: Vergrößerung des Targets

## 3.2 Aktueller Stand des Axismappers

Das Programm RacoonKinematicsCustomLogic.cs im Axismapper erstellt zuerst eine Klasse für „RacoonKinematicsCustomLogic“ und initialisiert anschließend Daten. Der Code besteht aus mehreren Funktionen, die jedoch nicht alle verwendet werden. Zu den verwendeten Funktionen zählen:

- Public RacoonKinematicsCustomLogic()
- Public void SetRotatorModeNormal()
- Public void SetRotatorModeTurn()
- Public unsafe override void CallbackFunctionProcessStates()
- Public void UpdateAxisPositionVelocity()
- Void CompareTransforms()
- Public void Kine()

In den Funktionen „SetRotatorModeTurn“ und „SetRotatorModeNormal“ werden einige Achsen an- bzw. ausgeschalten und stammen aus der vorherigen Berechnung der Kinematik. Diese werden nicht mehr benötigt, da bei der neuen Beschreibung der Kinematik alle Achsen zum Einsatz kommen.

Bisher wird die Kinematik mit einer „InvertLogic“, der Funktion public void Kine() und einer Abfrage nach dem Modus (SetRotatorModeTurn oder SetRotatorModeNormal) berechnet. Diese werden als Ergebnis dieser Arbeit durch ein neues Konzept zur Berechnung der Inversen Kinematik ersetzt.

# 4 Einführung in die Robotik

Roboterbewegungen implizieren, per Definition, Bewegungen von Teilen und Werkzeugen im Raum und eine Art von Mechanismus. Das führt logischerweise zu einer Notwendigkeit Positionen und Orientierung von Teilen bzw. Werkzeugen und des gesamten Mechanismus‘ darzustellen. Für eine mathematische Beschreibung und Veränderung von Position und Orientierung braucht man Koordinatensysteme und Regeln. (Craig 2014: 19)

Dieses Kapitel bildet die Basis für alle notwendigen Berechnungen, Darstellungen und Transformationen, um die Kinematik des RACOON-Labs zu beschreiben. Es beschreibt die direkte und inverse Kinematik sowie deren Unterschiede und Anwendungsbereiche. Für die Berechnungen der inversen Kinematik sind insbesondere Jakobimatrizen und die Planung der Trajektorien von Interesse.

## 4.1 Darstellungsweisen und Transformationen

Zu Beginn werden Darstellungsweisen der Position und Orientierung eingeführt, um die Lage eines Körpers im Raum eindeutig festlegen zu können. Die Änderung der Lage des Körpers lässt sich mit Transformationsmatrizen beschreiben, die sich relativ zu einem körperfesten oder absolut zu einem inertialen Koordinatensystem beziehen können.

### 4.1.1 Position, Orientierung und Koordinatensysteme

Eine Beschreibung des Roboters wird verwendet, um Attribute von Objekten, mit denen ein Roboter arbeitet, zu spezifizieren. Diese Objekte sind Werkzeuge, Teile des Roboters und der Roboter selbst. Dafür muss man Position und Orientierung relativ zu einem anderen Objekt oder absolut im inertialen Koordinatensystem darstellen können. Nach Einführung eines Koordinatensystems kann jeder Punkt im Universum mit einem 3 x 1 Positionsvektor lokalisiert werden. Da oft mehrere Koordinatensysteme eingeführt werden, muss der Vektor eindeutig gekennzeichnet sein, zu welchem Koordinatensystem er referenziert wird. (Craig 2014: 20)

Für diese Arbeit bezeichnet ein voran- und tiefgestellter Index das Referenzsystem. Der Index „0“ bezieht sich dabei auf das Inertialsystem ($B_0$) und der Index „i“ relativ zu einem körperfesten KOS ($B_i$). Ein Punkt relativ zu einem Koordinatensystem i wird wie folgt dargestellt:

$$ {}_{i}r_P = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \qquad \textbf{( 4–1 )} $$

In vielen Anwendungen ist nicht nur die Position, sondern auch die Orientierung einzelner Manipulatoren bzw. des Endeffektors (das letzte Glied in der Kette) von zentraler Bedeutung. Wenn der Vektor ${}_{i}r_P$ den Ort des Endeffektors, z.B. eines Greifarms, angibt, ist der Ort ohne die Orientierung nicht eindeutig festgelegt. Mit einer ausreichenden Anzahl an Gelenken könnte der Punkt P willkürlich orientiert sein, ohne seine Position im Raum zu verändern. Um das zu vermeiden, befestigt man ein

Koordinatensystem an den Punkt P, welches dann relativ zum Referenzsystem beschrieben wird (vgl. Abb. 4–1).

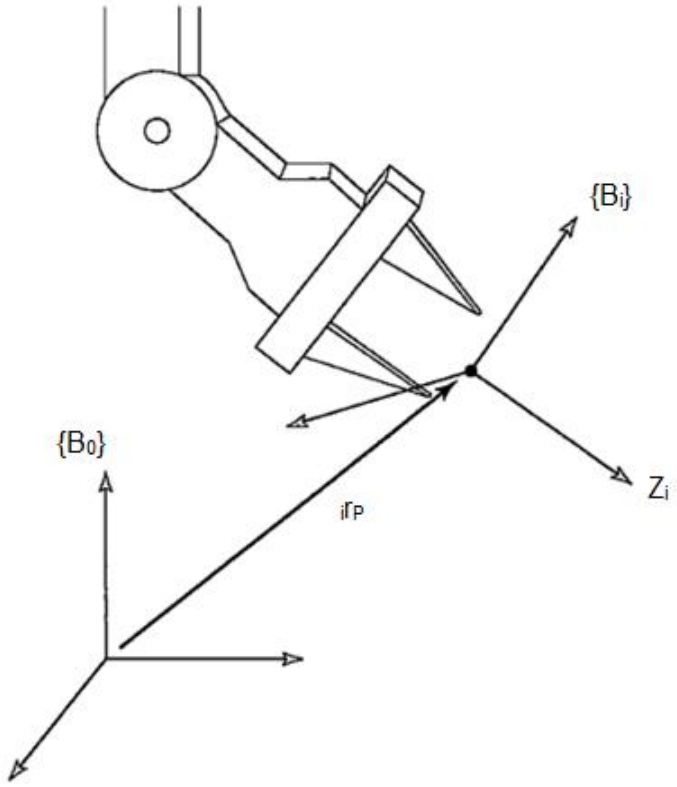



Abb. 4–1: Position und Orientierung eines Punktes (angelehnt an Abb. in (Craig 2014: 21))

Das körperfeste Koordinatensystem wird durch eine 3 x 3 Matrix mit 3 Einheitsvektoren beschrieben, die im Referenzsystem dargestellt werden. Die Rotationsmatrix $A_{i0} \in R^{3x3}$ transformiert Größen von einer Darstellung im $B_0$ in eine Darstellung ins $B_i$-System:

$$ {}_i r_P = A_{i0} \cdot {}_0 r_P \qquad \textbf{( 4–2 )} $$

### 4.1.2 Räumliche Transformationen

Wenn eine Basis durch Rotation um eine der drei Achsen x, y oder z aus einer anderen hervorgeht, nennt man diese Elementardrehung:

$$ A_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) \\ 0 & -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \qquad \textbf{( 4–3 )} $$

$$ A_y(\beta) = \begin{pmatrix} \cos(\beta) & 0 & -\sin(\beta) \\ 0 & 1 & 0 \\ \sin(\beta) & 0 & \cos(\beta) \end{pmatrix} \qquad \textbf{( 4–4 )} $$

$$A_z(\gamma) = \begin{pmatrix} \cos(\gamma) & \sin(\gamma) & 0 \\ -\sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \textbf{( 4–5 )}$$

Da die Einheitsvektoren die Spalten der Drehmatrix bilden, entspricht die Inverse also der Transponierten, was für kinematischen Berechnungen oft ausgenutzt wird: (Pfeiffer and Schindler 2014: 11-3) (Craig 2014: 25)

$$A_{ij}^{-1}=A_{ij}^{T} \quad \textbf{( 4–6 )}$$

Die bekanntesten Darstellungen von hintereinander geschalteten Elementardrehungen sind die Euler(ZYX)-, Euler(ZYZ)- und Roll/-Nick-/Gierwinkel. Für die Darstellung in Eulerwinkel(ZYX) dreht man zuerst um einen Winkel α um die z-Achse, anschließend um einen Winkel β um die gedrehte y-Achse und zuletzt um einen Winkel γ um die gedrehte x-Achse:

$$A_{Euler(ZYX)}=(A^{T}_{x}(\alpha)A^{T}_{y}(\beta)A^{T}_{z}(\gamma))^{T}=A_z(\gamma)A_y(\beta)A_x(\alpha) \quad \textbf{( 4–7 )}$$

Für die Eulerwinkel-Darstellung erfolgt zuerst eine Drehung um die z-Achse, dann um die y-Achse und schließlich um die bereits gedrehte z-Achse (Pfeiffer and Schindler 2014: 11–3):

$$A_{Euler(ZYZ)}=A_z(\gamma)A_y(\beta)A_z(\alpha) \quad \textbf{( 4–7 )}$$

Bei der Roll-/Nick-/Gierdrehung erfolgen die Rotationen um die fixen Achsen in der Reihenfolge X-Y-Z:

$$A_{Rollen/Nicken/Gieren}=A_z(\gamma)A_y(\beta)A_x(\alpha) \quad \textbf{( 4–8 )}$$

Für Euler(ZYX) ergibt sich die selbe Rotationsmatrix wie für Rollen/Nicken/Gieren, deshalb kann man die selbe Formeln zur Berechnung der Winkel verwenden (Gl. 5–10 bis 5–12). Für eine detaillierte Herleitung sei auf (Craig 2014: 42–5) verwiesen.

Für die Denavit-Hartenberg-Transformation, die in 4.1.4 näher erläutert wird, benötigt man nur $A_z(\gamma)$ und $A_x(\alpha)$.

Das inverse Problem, bei dem eine Transformationsmatrix gegeben ist und man die Rotationswinkel herausfinden möchte, ist ebenfalls oft von Interesse. Da für diese Arbeit die DH-Transformation verwendet wird und die Reihenfolge der Transformationen mit denen der Roll/Nick/Gierdrehung übereinstimmt, wird hierfür die Berechnung der Winkel gezeigt. Da es bei DH keine Rotation um y gibt, kann diese mit einer Rotation von 0° berücksichtigt werden.

Für die Berechnung verwendet man die Atan2(y,x)-Funktion, die die Quadranten der Winkel mit berücksichtigt. Für eine Transformationsmatrix A ergibt sich: (Craig 2014: 42–3)

$$A = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \quad \textbf{( 4–9 )}$$

$$\beta=\text{Atan2}(-r_{31},\sqrt{r_{11}^2+r_{21}^2}) \quad \textbf{( 4–10 )}$$

$$\alpha=\text{Atan2}(r_{21}/\cos(\beta),r_{11}/\cos(\beta)) \quad \textbf{( 4–11 )}$$

$$\gamma=\text{Atan2}(r_{32}/\cos(\beta),r_{33}/\cos(\beta)) \quad \textbf{( 4–12 )}$$

### 4.1.3 Denavit-Hartenberg-Konvention

Die Denavit-Hartenberg-Konvention (DH-Konvention) ist eine in der Robotik weit verbreitete Methode, um die relative Lage zweier Körper zueinander eindeutig zu beschreiben. Diese Parametrisierung umfasst nicht nur Drehgelenke, sondern auch Schubgelenke. Ein großer Vorteil dieser Konvention, ist die Tatsache, dass man für die eindeutige Beschreibung der Kinematik nur vier Parameter benötigt. Im Allgemeinen sind zur Beschreibung der relativen Lage und Orientierung zweier Koordinatensysteme sechs Parameter notwendig. Da nur Dreh- und Schubgelenke dargestellt werden, ermöglicht das die Verringerung der erforderlichen Parameter. Ebenso wird auf die freie Wahl der Lage und Orientierung der Koordinatensysteme relativ zu den bewegten Körpern verzichtet. Aufgrund der weiten Verbreitung dieser Konvention ist der Informationsaustausch sehr leicht, jedoch ist zu beachten, dass es in der Literatur geringe Unterschiede gibt. Im Folgenden wird die Konvention nach Craig verwendet.

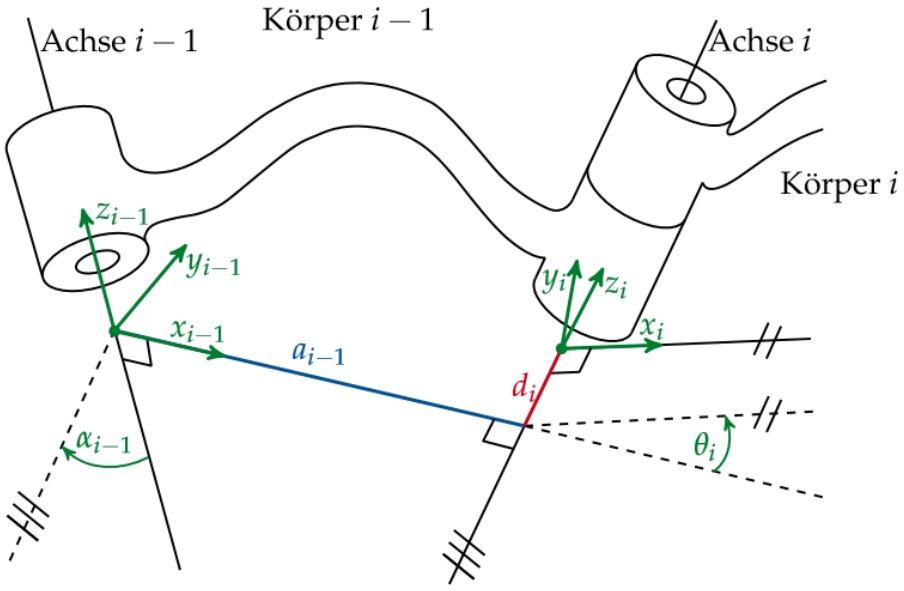



Abb. 4–2: Die Denavit-Hartenberg-Parameter (Abbildung nach (Craig 2014))

Das prinzipielle Vorgehen zur Festlegung der Koordinatensysteme (Abb. 4–2) :

Zuerst erfolgt die Nummerierung der Körper von 0 bis N, wobei der Körper 0 fix mit der Umgebung verbunden ist. Die Koordinatenachse $z_i$ ist identisch mit der Bewegungsachse i. Dabei ist die Richtung der Koordinatenachse durch die positive Dreh- oder Verschiebungsrichtung des Freiheitsgrads vorgegeben. Die

Koordinatenachse $x_i$ bildet eine senkrechte Verbindungslinie von $z_i$ nach $z_{i+1}$. Die Koordinatenachse $y_i$ ergibt sich aus $x_i$ und $z_i$ zu einem Rechtssystem. Falls die Wahl der Koordinatensysteme nicht eindeutig ist, sollten die Achsen so gewählt werden, dass viel DH-Parameter zu 0 werden.

Die Denavit-Hartenberg-Parameter:

- $a_{i-1,i}$ = Distanz von $z_{i-1}$ nach $z_i$ in Richtung $x_i$
- $\alpha_{i-1,i}$ = Winkel zwischen $z_{i-1}$ und $z_i$, positiv gegen den Uhrzeigersinn in positiver $x_i$ –Richtung
- $d_{i-1,i}$ = Distanz von $x_{i-1}$ nach $x_i$ in Richtung $z_i$
- $\theta_{i-1,i}$ = Winkel zwischen $x_{i-1}$ und $x_i$, positiv gegen den Uhrzeigersinn in positiver $z_{i-1}$-Richtung

(Craig 2014: 68–9) (Shigley and Uicker 1995)

### 4.1.4 Denavit-Hartenberg-Transformation

Die relative Verschiebung sowie Verdrehung zweier Koordinatensysteme kann durch eine homogene Transformationsmatrix $D_{ji} \in R^{4x4}$ beschrieben werden. $D_{ji}$ gibt die Transformation einer Basis i relativ zur Basis j an.

Ein Punkt kann zur Basis i durch den Ortsvektor ${}_ir_{iP}$ oder durch den Ortsvektor ${}_jr_{jP}$, relativ zur Basis j, dargestellt werden. Die Umrechnung zwischen beiden Darstellungen lässt sich mit der DH-Transformation realisieren:

$$ {}_jr_{jP} = D_{ji} \cdot {}_ir_{iP} \qquad \textbf{( 4–13 )} $$

Die DH-Transformation kann, ähnlich wie bei räumlichen Drehungen, durch vier elementare homogene Transformationen zusammengesetzt werden. Diese entstehen entweder durch Drehung oder Verschiebung entlang einer der Koordinatenachsen:

1. Rotation $\theta_i$ um $z_i$: $D_1 = \begin{pmatrix} A_z(\theta_i) & 0 \\ 0 & 1 \end{pmatrix}$
2. Translation $d_i$ entlang $z_i$: $D_2 = \begin{pmatrix} E & e_z d_i \\ 0 & 1 \end{pmatrix}$
3. Translation $a_{i-1}$ entlang $x_{i-1}$: $D_3 = \begin{pmatrix} E & e_z a_{i-1} \\ 0 & 1 \end{pmatrix}$
4. Rotation $\alpha_{i-1}$ um $x_{i-1}$: $D_4 = \begin{pmatrix} A_x(\alpha_{i-1}) & 0 \\ 0 & 1 \end{pmatrix}$

Die resultierende homogene Transformation (s=sin, c=cos) (Craig 2014: 75):

$$D_{i-1,i}=D_4D_3D_2D_1=\begin{pmatrix} A_x^T(\alpha_{i-1})A_z^T(\theta_i) & A_x^T(\alpha_{i-1})(a_{i-1}e_x+d_ie_z) \\ 0 & 1 \end{pmatrix}$$

$$=\begin{pmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1}d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad \textbf{( 4–14 )}$$

Folgende Beziehungen gelten, wenn man die Jakobimatrix in eine translatorische ($J_T$) und eine rotatorische ($J_R$) Jakobimatrix aufteilt, wobei ω die Winkelgeschwindigkeiten und v die translatorischen Geschwindigkeiten bezeichnen:

$$\omega=J_R\cdot\dot{q} \text{ und } v=J_T\cdot\dot{q} \quad \textbf{( 4–15 )}$$

Zuerst werden die relativen Größen für Drehgelenke hergeleitet, anschließend die für Schubgelenke.

**Drehgelenke:**

Für Drehgelenke ist di eine feste Größe und $\theta_i=q_i$ bezeichnet den dazugehörigen Freiheitsgrad. Ausgehend von der DH-Transformation (Gl. 4−14) folgen die Drehmatrix und die Relativverschiebung:

$$r_{i-1,i}=A_x^T(\alpha_{i-1})(a_{i-1}e_x+d_ie_z)=\text{konstant.}$$
$$A_{i,i-1}=A_z(q_i)A_x(\alpha_{i-1}) \quad \textbf{( 4–16 )}$$

Die übrigen benötigten Größen werden durch Differentiation berechnet:

$$v_{i-1,i}=\omega_{i-1}\times r_{i-1,i}=\tilde{r}_{i-1,i}^T\cdot\omega_{i-1}$$
$$\omega_{i-1,i}=e_z\dot{q}_i \quad \textbf{( 4–17 )}$$

$$J_{R,rel,i,[m,n]}=1, \text{ für } m=3 \text{ und } n=q \quad \textbf{( 4–18 )}$$

$$J_{T,rel,i}=A_{i,i-1}\cdot(\tilde{r}_{i-1,i}^T\cdot J_{R,i-1}) \quad \textbf{( 4–19 )}$$

**Schubgelenke:**

Für Schubgelenke entspricht $d_i$ dem Freiheitsgrad $q_i$, während $\theta_i$ eine feste Größe ist. Aus Gl. 4−14 erhält man:

$$r_{i-1,i}=A_x^T(\alpha_{i-1})(a_{i-1}e_x+d_ie_z)\neq\text{konstant}$$
$$A_{i,i-1}=A_z(\theta_i)A_x(\alpha_{i-1})=\text{konstant.} \quad \textbf{( 4–20 )}$$

Für $v_{i-1,i}$ und $\omega_{i-1,i}$ folgen:

$$v_{i-1,i}=A_x^T(\alpha_{i-1})e_z\dot{q}_i+\omega_{i-1}\times r_{i-1,i}$$
$$\omega_{i-1,i}=0 \quad \textbf{( 4–21 )}$$

Daraus ergeben sich die Jakobimatrizen:

$$J_{R,rel,i}=0 \qquad \textbf{( 4–22 )}$$

$$J_{T,rel,i}=A_{i,i-1}\cdot(\tilde{r}_{i-1,i}\cdot J_{R,i-1})+A_{i,i-1}\cdot A_x^T(\alpha_{i-1})\cdot e_z \qquad \textbf{( 4–23 )}$$

$$\text{Mit } \tilde{r}_{vi}=\begin{pmatrix} 0 & -r_{vi,z} & r_{vi,y} \\ r_{vi,z} & 0 & -r_{vi,x} \\ -r_{vi,y} & r_{vi,x} & 0 \end{pmatrix} \qquad \textbf{( 4–24 )}$$

$\tilde{r}_{vi}$ wird aus dem Verschiebungsvektor $r_{vi}$ gebildet, damit die Dimensionen der Matrizen wieder zueinander kompatibel sind.

## 4.2 Direkte Kinematik

Die direkte Kinematik hat das Finden der Lage und der Orientierung sowie die Geschwindigkeit und Beschleunigung des Endeffektors zur Aufgabe, wenn die Geometrie jeder Komponente und die Position jedes Aktuators, die die Freiheitsgrade steuern, bekannt sind (Shigley and Uicker 1995: 425). Nach Bestimmung der DH-Parameter für einen Robotermanipulator, werden mit Hilfe der Gleichungen 4-2 bis 4-5 die Position und Ausrichtung der einzelnen Gelenke relativ zum Vorgänger ermittelt. Zu berücksichtigen ist dabei der systematische Durchlauf der kinematischen Kette, da nachfolgende Glieder durch eine Veränderung des Freiheitsgrades des vorherigen Glieds ebenfalls beeinflusst werden. Dabei spielt es keine Rolle, ob der Roboter redundant ist oder nicht, da alle Variablen bekannt sind. Die direkte Kinematik findet daher Anwendung bei der Initialisierung des Roboters, um feststellen zu können wie weit der Roboter, ausgehend von seiner Ruheposition, von seiner Start- bzw. Endposition entfernt ist.

## 4.3 Inverse Kinematik

Die inverse Kinematik beschäftigt sich mit der Bestimmung der Gelenkvariablen, deren Geschwindigkeit und Beschleunigung, bei gegebener Position, Geschwindigkeit und Ausrichtung des Endeffektors. Diese Aufgabe stellt sich in der Robotik von größerer Bedeutung als die direkte Kinematik heraus, da bei den meisten Problemen der Roboter eine bestimmte Position im Raum und keine bestimmten Gelenkwinkelstellungen erreichen soll (Craig 2014: 101). Darüber hinaus kommt ein zeitlicher Aspekt hinzu, wenn der Roboter einen bestimmten Pfad wandern soll. Das führt zu einer zeitlichen Abhängigkeit der Transformationen und der Gelenkfreiheitsgraden, welche bei der direkten Kinematik noch keine Rolle gespielt hat (Shigley and Uicker 1995: 429).

Ein Set mit einer minimalen Anzahl an Parametern zur eindeutigen Beschreibung der Lage und Orientierung eines Punktes, nennt man Minimalkoordinaten **q**, welche oft mit den Freiheitsgraden des Systems zusammenfallen (Pfeiffer and Reithmeier 1987: 25). Der Arbeitsraum ist der Raum, den der Endeffektor erreichen kann. D.h., wenn es eine Lösung geben soll, dann muss ein Punkt, den der Roboter erreichen soll, innerhalb

des Arbeitsraums liegen. Daraus ergeben sich die Arbeitsraumkoordinaten **w**, die die Lage und Orientierung des Endeffektors im Arbeitsraum an Hand einer minimalen Anzahl an Parametern bestimmen (Craig 2014: 102).

Daraus lassen sich Aussagen über die Lösbarkeit eines Problems treffen (Siciliano et al. 2009: 91):

- Es gibt genau eine Lösung (nicht bei redundanten Robotern)
- Es gibt keine Lösung
- Es gibt mehrere oder unendlich viele Lösungen

Die Anzahl an Lösungen hängt zum einen von der Anzahl an Gelenken und zum anderen von den Parametern ab. Je mehr Parameter ungleich null sind, desto mehr Lösungen existieren (Craig 2014: 104–5). Im Allgemeinen können die Dimension des Arbeitsraumes **w** und die Dimension der Minimalkoordinaten **q** unterschiedlich sein, was für das RACOON-Lab zutrifft. Zur Beschreibung der Lage und Orientierung werden drei Ortskoordinaten und sechs Winkelkoordinaten, jeweils drei für den Chaser und jeweils drei für das Target, benötigt. Das System besitzt jedoch elf Freiheitsgrade, was zu einer Redundanz führt und sich damit die Existenz von mehreren Lösungen im Vorhinein bestätigen lässt. Das Vorgehen zur Bestimmung einer Lösung wird im Kapitel 5.2 behandelt.

## 4.4 Planung der Trajektorien

Die Planung der Trajektorien umfasst die Veränderung der Position, Geschwindigkeit und Beschleunigung über die Zeit für jeden Freiheitsgrad. Um die Beschreibung der Manipulatorbewegung einfach zu halten, muss es dem Nutzer möglich sein, die Bewegung ohne komplizierte Formeln darzustellen und dem System die Details ausarbeiten lassen (Craig 2014: 201).

Wenn ein Roboter den Endeffektor von einer inertialen Position zu einer Zielposition bewegen soll, erlaubt die inverse Kinematik die Bestimmung der Minimalkoordinaten zum Zeitpunkt $t_0$ und zum Endzeitpunkt $t_f$. Nun kann man mit Hilfe von Randbedingungen und einem kubischen Polynom (Gl. 4-15) eine Trajektorie für jeden Freiheitsgrad vom Zeitpunkt $t_0$ bis $t_f$ generieren. Beispielhaft sind mögliche Pfade für einen Winkel θ in Abb. 4–3 zu sehen. (Craig 2014: 203–4)

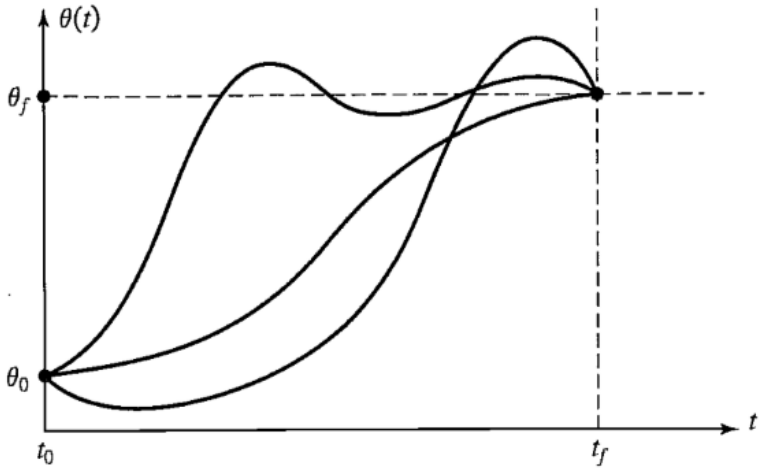



Abb. 4–3: Mögliche Pfade für einen Winkel θ (Craig 2014: 204)

$$\theta(t)=a_0+a_1t+a_2t^2+a_3t^3 \quad \textbf{( 4–15 )}$$

$$\dot{\theta}(t)=a_1+2a_2t+3a_3t^2 \quad \textbf{( 4–16 )}$$

$$\theta(0)= \theta_0,\ \theta(t_f)=\theta_f,\ \dot{\theta}(0)=0,\ \dot{\theta}(t_f)=0 \quad \textbf{( 4–17 )}$$

Wenn die Start- und Endgeschwindigkeiten eines Gelenks zu null gewählt werden, ergeben sich aus den Gl. 4-14 bis 4-15 und den Randbedingungen aus 5-16 die Parameter $a_i$ wie folgt: (Craig 2014: 204–5)

$$\begin{aligned} a_0&=\theta_0 \\ a_1&=0 \\ a_2&=\frac{3}{t_f^2}(\theta_f-\theta_0) \\ a_3&=-\frac{2}{t_f^3}(\theta_f-\theta_0) \end{aligned} \quad \textbf{( 4–18 )}$$

Wenn die Geschwindigkeiten zum Zeitpunkt $t_0$ und $t_f$ bekannt sind, kann man die Geschwindigkeitsrandbedingungen, die hier null sind, durch jene bekannten Geschwindigkeiten ersetzen. Daraus ergeben sich für ein kubisches Polynom folgende Koeffizienten:

$$\begin{aligned} a_0&=\theta_0 \\ a_1&=\dot{\theta}_0 \\ a_2&=\frac{3}{t_f^2}(\theta_f-\theta_0) - \frac{2}{t_f}\dot{\theta}_0-\frac{1}{t_f}\dot{\theta}_f \\ a_3&=-\frac{2}{t_f^3}(\theta_f-\theta_0) + \frac{1}{t_f^2}(\dot{\theta}_f+\dot{\theta}_0) \end{aligned} \quad \textbf{( 4–19 )}$$

# 5 Kinematische Beschreibung des RACOON-Lab

Mit Hilfe der mathematischen Grundlagen und den technischen Parametern des RACOON-Labs kann nun die Kinematik bestimmt werden. Für die Modellierung benötigt man die Randbedingungen der Freiheitsgrade, die Systemmaße, die kinematische Kette und die Umrechnung der Orbitdaten. Da das System mehr Freiheitsgrade als Arbeitsraumkoordinaten besitzt, muss der allgemeine Ansatz zur Lösung erweitert werden. Nach der mathematischen Formulierung des Problems, werden zwei mögliche Lösungsvarianten vorgestellt.

## 5.1 Systemmodellierung

Die Systemmodellierung ist ein essentieller Bestandteil bei der Bestimmung der inversen Kinematik. Es gibt verschiedene Möglichkeiten die Kinematik zu beschreiben oder die Winkel und Transformationsmatrizen zu berechnen. Eine einfache Methode ist die Beschreibung mit Denavit-Hartenberg-Parametern, die für die relative Lage zweier Gelenke nur vier Variablen benötigt. Die Freiheitsgrade sind immer entlang oder um die z-Achse, wobei diese nicht zwingend mit den Arbeitsraumkoordinaten übereinstimmen müssen. Die Arbeitsraumkoordinaten werden vom Benutzer festgelegt und stellen die Lage und Orientierung des Endeffektors im inertialen System dar. Beispielsweise kann aufgrund der Beschreibung ein Winkel um π größer oder kleiner sein, was einen direkten Vergleich der Freiheitsgrade mit den Arbeitsraumkoordinaten teilweise erschwert. Um das zu vereinfachen, dreht man das inertiale Koordinatensystem bis der Freiheitsgrad mit der z-Achse übereinstimmt. Dadurch entstehen mehrere „Sets“ an DH-Parametern, da mehrere Drehungen notwendig sein können, bevor die Achsen zusammenfallen. Ebenso sollte das letzte System wieder mit dem inertialen Koordinatensystem enden. Die Transformationsmatrizen werden dann genauso gebildet als wäre die Beschreibung direkt mit DH erfolgt. Der Ursprung des inertialen Koordinatensystems befindet sich für diese Anwendung im inneren des Targets, weil sich dieses aufgrund der Bauweise nicht translatorisch bewegen kann und deshalb nur die Rotationen betrachtet werden müssen. Alle Translationen werden relativ vom Target zum Chaser berechnet. Eine andere Wahl des Ursprungs ist auch möglich, da die Kinematik nicht von dessen Lage abhängt.

Die Tab. 5–1 und Tab. 5–2 zeigen alle DH-Parameter für das RACOON-Lab auf. Parameter a bedeutet eine Translation in Richtung der lokalen x-Achse, α eine Rotation um die lokale x-Achse, θ eine Rotation um die lokale z-Achse und d eine Translation um die lokale z-Achse. Aus Tab. 5–3 und Tab. 5–4 können mit Hilfe der Koordinatensysteme die DH-Parameter nachvollzogen werden.

Tab. 5–1: DH-Parameter für den Chaser

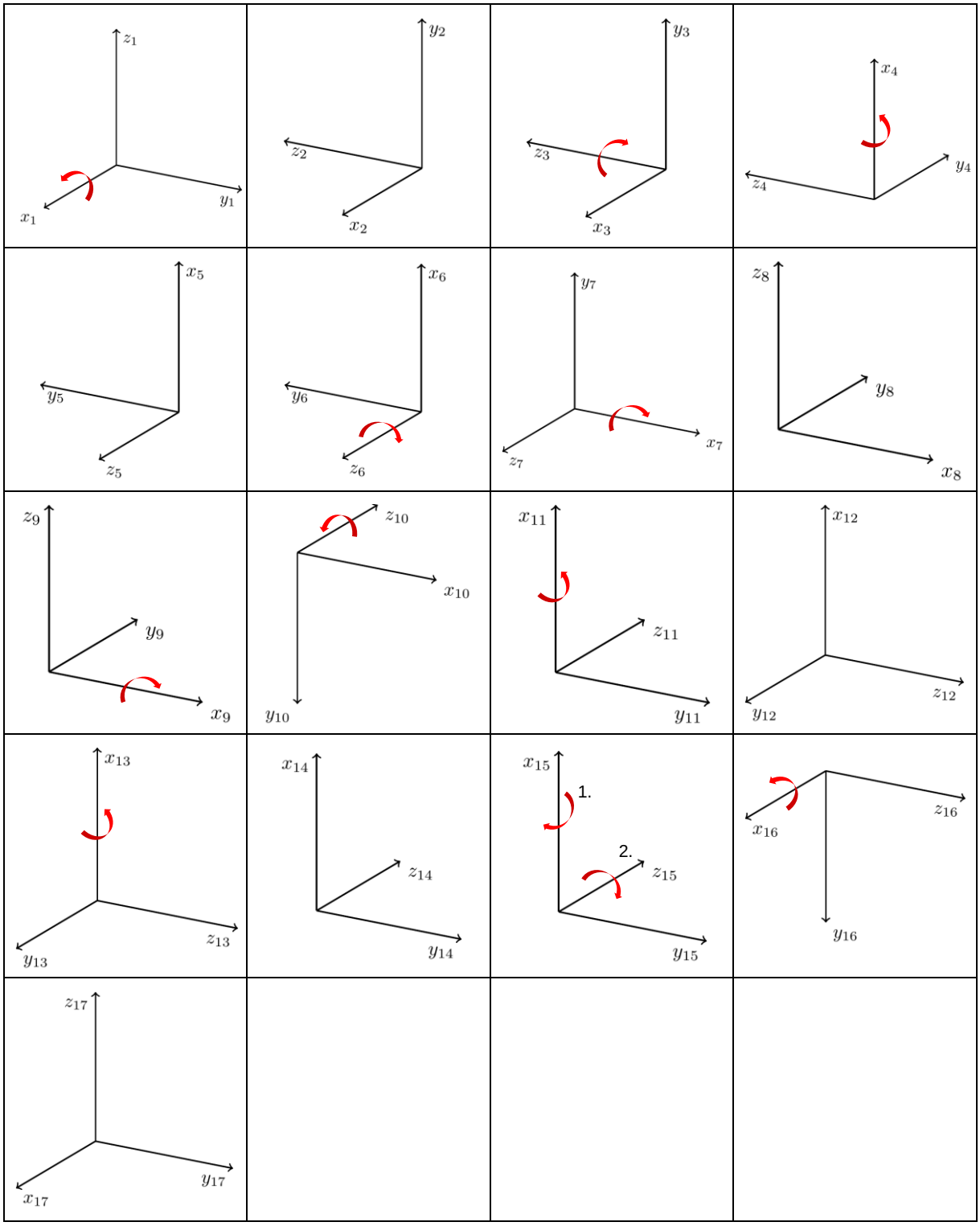
| Nummerierung | a | α | θ | d |
|---|---|---|---|---|
| **1** | 0 | 0 | 0 | $q_1$ |
| **2** | 0 | π/2 | 0 | 0 |
| **3** | 0 | 0 | 0 | $q_2$ |
| **4** | 0 | 0 | π/2 | 0 |
| **5** | 0 | π/2 | 0 | 0 |
| **6** | 0 | 0 | 0 | $q_3$ |
| **7** | 0 | 0 | - π/2 | 0 |
| **8** | 0 | - π/2 | 0 | 0 |
| **9** | 0 | 0 | $q_4$ | 0 |
| **10** | 0 | -π/2 | 0 | xy |
| **11** | 0 | 0 | - π/2 | 0 |
| **12** | 0 | - π/2 | 0 | 0 |
| **13** | 0 | 0 | $q_5$ | 0 |
| **14** | 0 | π/2 | 0 | 0 |
| **15** | 0 | 0 | $q_6$ | 0 |
| **16** | 0 | - π/2 | π/2 | 0 |
| **17** | 0 | π/2 | 0 | 0 |

Tab. 5–2: DH-Parameter für das Target

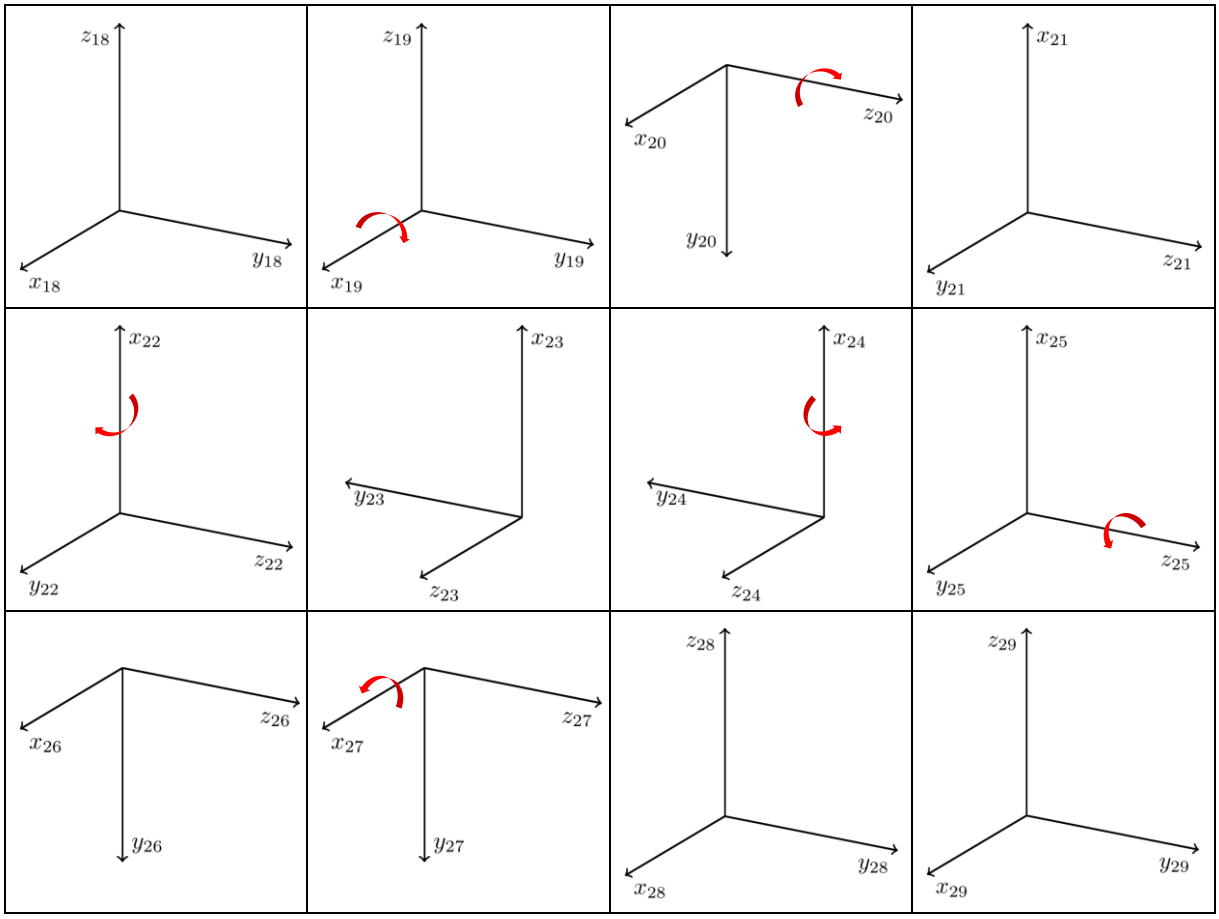
| Nummerierung | a | α | θ | d |
|---|---|---|---|---|
| **18** | 0 | 0 | 0 | XY |
| **19** | 0 | 0 | $q_7$ | 0 |
| **20** | 0 | - π/2 | 0 | 0 |
| **21** | 0 | 0 | -π/2 | 0 |
| **22** | 0 | 0 | $q_8$ | 0 |
| **23** | 0 | - π/2 | 0 | 0 |
| **24** | 0 | 0 | $q_9$ | 0 |
| **25** | 0 | π/2 | 0 | 0 |
| **26** | 0 | 0 | π/2 | 0 |
| **27** | 0 | 0 | $q_{10}$ | 0 |
| **28** | 0 | π/2 | 0 | 0 |
| **29** | 0 | 0 | $q_{11}$ | 0 |

Die Koordinatensysteme in Tab. 5–3 und Tab. 5–4 zeigen den Zustand nach Anwendung der Rotation bzw. Translation an. Wenn ein Freiheitsgrad auftritt, bleibt das Koordinatensystem in der Abbildung unverändert (vgl. $x_2y_2z_2$ und $x_3y_3z_3$). Die roten Pfeile deuten die Rotation um die jeweilige Achse an, um die für die Erzeugung des nächsten Koordinatensystems gedreht wird. Beispielsweise dreht man das KOS 1 um π/2 (vgl. Tab. 5–1: Nummerierung: 2, α= π/2) um die x-Achse damit man das KOS 2 erhält. Nach diesem Schema geht man so lange vor, bis alle Freiheitsgrade berücksichtigt wurden und das Endkoordinatensystem wieder dem Inertialsystem entspricht. Dieses Vorgehen entspricht nicht komplett der DH-Parametrisierung, dennoch können diese Parameter nach dem Schema ausgewertet und die Transformationsmatrizen gebildet werden. Aufgrund der Komplexität des Systems ist die Abarbeitung Schritt für Schritt übersichtlicher.

Tab. 5–3: Koordinatensysteme für den Chaser nach Transformation mit DH



Tab. 5–4: Koordinatensysteme für das Target nach Transformation mit DH




Aus der Parametrisierung erfolgt die Berechnung der Relativmatrix:

$$A_{i,i-1}=A_z(\theta_i)A_x(\alpha_i) \qquad \textbf{( 5–1 )}$$

Diese unterscheidet sich nur durch die Drehung des Winkels $\alpha_i$ um die x-Achse und nicht wie bei DH um $\alpha_{i-1}$.

In Kapitel 3.1 wurden bereits die kinematischen Ketten für das Target und den Chaser identifiziert. Das RACOON-Lab besteht aus zwei „Armen“, welche eine Baumstruktur bilden. Abb. 5–2 zeigt die Freiheitsgrade in ihrer richtigen Reihenfolge auf. Dabei stellt der linke Ast die Freiheitsgrade des Chasers und der rechte Ast die des Targets dar. Zum einfacheren Verständnis des Systems ist eine Prinzipskizze hilfreich (vgl. Abb. 5–1). Das „C“ in dem das Target sich um die y-Achse drehen kann, wird hier mit einem roten Teilkreis angedeutet und auf der linken Seite befindet sich der Chaser. Zur besseren Vergleichbarkeit ist die Nummerierung der Achsen identisch mit der Nummerierung aus Tab. 5–1 und Tab. 5–2.

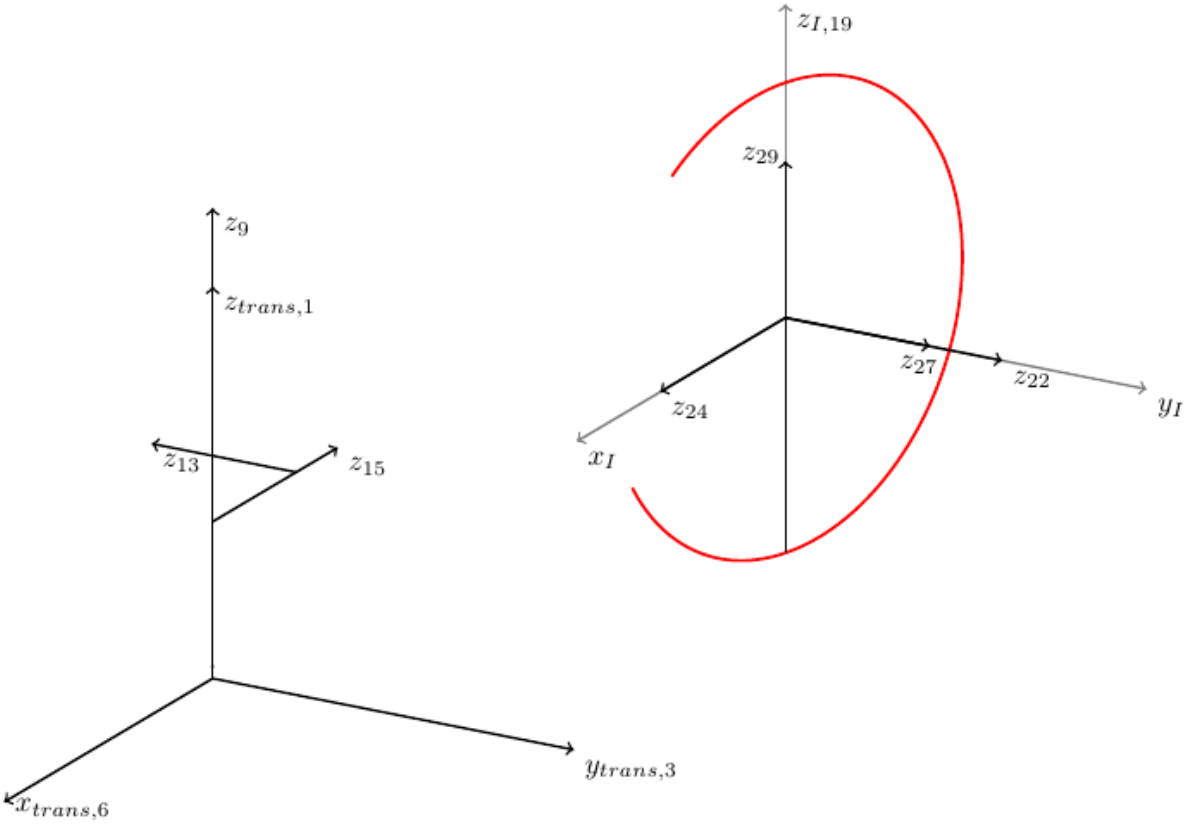



Abb. 5–1: Prinzipskizze des RACOON-Labs mit den Freiheitsgraden und dem rot angedeuteten „C“

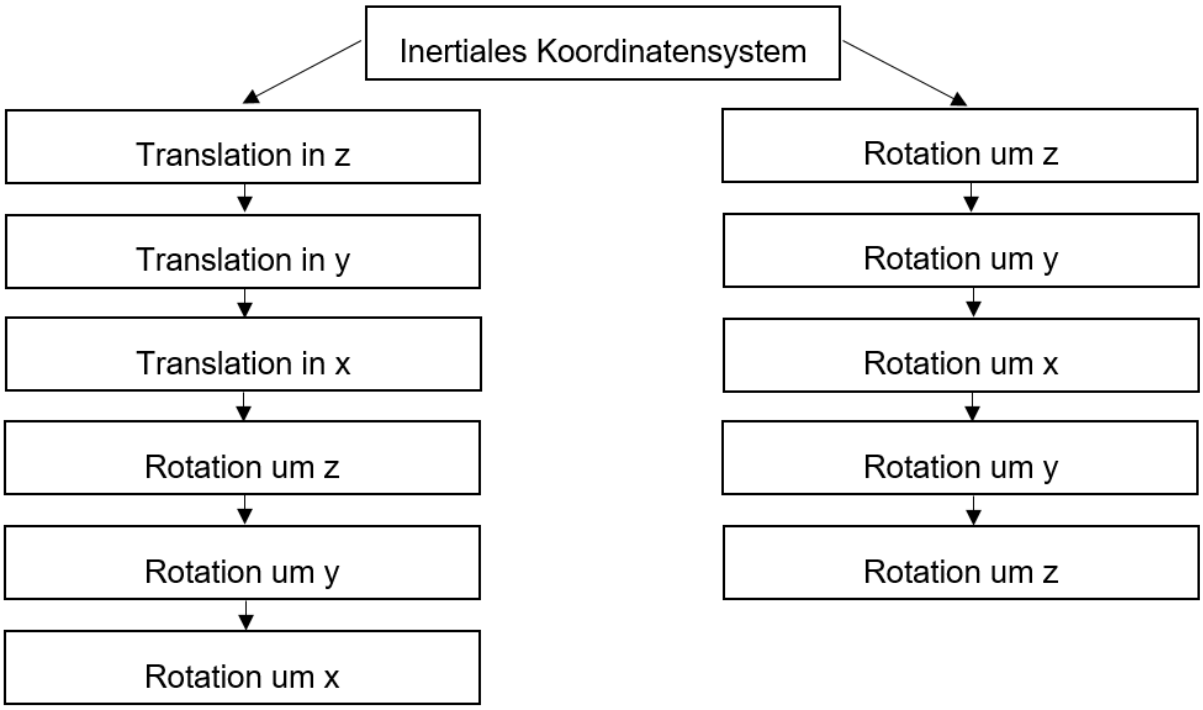



Abb. 5–2: Baumstruktur der Freiheitsgrade des RACOON-Lab (Links: Chaser; Rechts: Target)

Nach der Modellierung der Kinematik werden die Arbeitsraumkoordinaten definiert. Hier bietet es sich aufgrund der Wahl der Freiheitsgrade an den Arbeitsraum wie folgt zu bestimmen:

$$w = \begin{bmatrix} x \\ y \\ z \\ \gamma_{Chaser} \\ \beta_{Chaser} \\ \alpha_{Chaser} \\ \gamma_{Target} \\ \beta_{Target} \\ \alpha_{Target} \end{bmatrix} \quad \textbf{( 5–2 )}$$

Die Winkel γ, β und α werden mit Rollen/Nicken/Gieren jeweils aus der resultierenden Rotationsmatrix für den Chaser bzw. für das Target berechnet (Gl. (4 – 10) bis (4 – 12)). Wie in Kapitel 4.1.1 bereits erwähnt ist die Reihenfolge bei DH dieselbe wie bei Rollen/Nicken/Gieren, nur mit einer Rotation von β=0° um die y-Achse.

Der Relativvektor der beiden Satelliten ergibt sich aus der Vektoraddition beider Ortsvektoren. Da der Chaser nicht beliebig um das Target verfahren kann, wird der Relativvektor so gedreht, dass dieser im Arbeitsraum liegt. Damit die Ausrichtungen nach der Korrektur wieder übereinstimmen, werden diese ebenfalls gedreht. Der Relativvektor wird zur x-Achse der Anlage durch Rotationen um z bzw. y gedreht.

Tab. 5–5 stellt eine Verknüpfung der Achsenbezeichnungen des RACOON-Lab mit der Nummerierung der Freiheitsgrade des Systems dar. Die Reihenfolge ist nicht identisch, da die kinematische Kette nicht mit der bisherigen Nummerierung übereinstimmt.

Tab. 5–5: Achsenbezeichnungen

| Freiheitsgrad | Achsenbezeichnung des RACOON-Lab |
|---|---|
| $q_1$ | Servicer Linear Z |
| $q_2$ | Servicer Linear Y |
| $q_3$ | Servicer Linear X |
| $q_4$ | Servicer Rotation Z |
| $q_5$ | Servicer Rotation Y |
| $q_6$ | Servicer Rotation X |
| $q_7$ | Target C Arc Rotation Z |
| $q_8$ | Target C Sled Y |
| $q_9$ | Target Head Rotation X |
| $q_{10}$ | Target Head Rotation Y |
| $q_{11}$ | Target Head Rotation Z |

Im obigen Abschnitt wurden bereits die physischen Grenzen der Anlage angedeutet. Zum einen kann der Roboter nicht beliebig im Raum verfahren, zum anderen sind Rotationen durch die Mechanik des Systems begrenzt. Das ist zunehmend bei vielen Freiheitsgraden auf engem Raum der Fall, da sich die Motoren, Lager und andere Teilegegenseitig im Weg stehen. Tab. 5–6 zeigt die Grenzen für alle elf Freiheitsgrade des Systems auf. Diese müssen jedoch noch exakt vermessen werden, damit der Axismapper auch für das RACOON-Lab und nicht nur für die Softwaresimulation verwendet werden kann. Die Grenzen für Freiheitsgrad 4 bis 11 sind in [rad] angegeben. In Kapitel 5.3 wird dann eine Lösung für das Vermeiden von Systemgrenzen vorgestellt.

Tab. 5–6: Grenzen der Freiheitsgrade der Anlage

| Freiheitsgrad | Untere Grenze | Obere Grenze |
|---|---|---|
| $q_1$ | 0 m | 5 m |
| $q_2$ | −4 m | 4 m |
| $q_3$ | 0 m | 6 m |
| $q_4$ | −2π | 2π |
| $q_5$ | −2π | 2π |
| $q_6$ | −2π | 2π |
| $q_7$ | −2π | 2π |
| $q_8$ | −0.5935 | 3.5953 |
| $q_9$ | −π/4 | π/4 |
| $q_{10}$ | −π/6 | π/6 |
| $q_{11}$ | −2π | 2π |

## 5.2 Problemformulierung

Der erste Ansatz zur Formulierung des Problems erfolgt auf Positionsebene indem man die Ist-Position mit der Soll-Position vergleicht und sich durch Δq dieser annähert. Der Algorithmus stoppt, wenn Δq hinreichend klein ist. Die mathematische Formulierung lautet wie folgt:

$$w(q_0+\Delta q) \approx w(q_0)+\Delta q \cdot \delta w/\delta q \qquad \textbf{( 5–3 )}$$

$\delta w/\delta q$ bildet die Jakobimatrix $J_w$. Das größte Problem bei dieser Formulierung ist die Tatsache, dass die Dimensionen von w und q gleich sein müssen, damit nur redundante Roboter behandelt werden können. Die Konvergenz hängt auch von der Wahl des Startwerts ab. Bei einer guten Wahl konvergiert der Algorithmus schnell, bei einem schlechten Startwert divergiert der Algorithmus. Ein weiterer Nachteil ist das Fehlen einer Lösung, wenn die Jakobimatrix singulär ist.

Eine Verbesserung der Problemformulierung stellt die Betrachtung auf Geschwindigkeitsebene dar. Die Beziehungen auf Geschwindigkeitsebene sind,

anders als auf Lageebene, linear. Aus den Arbeitsraumgeschwindigkeiten und der inversen Jakobimatrix ergeben sich die Gelenkgeschwindigkeiten, aus denen man dann durch numerische Integration die Gelenkwinkel berechnen kann: (Siciliano et al. 2009: 132–3)

$$\dot{q}=J_w^{-1}\cdot\dot{w} \qquad \textbf{( 5–4 )}$$

Bei nicht-redundanten Robotern ist die Jakobimatrix nicht quadratisch, was zu einer Notwendigkeit für eine andere Berechnungsweise der inversen Jakobimatrix führt.

Das hierzu führende Verfahren nennt sich Resolved Motion Control Rate und wurde 1969 von Daniel E. Whitney entwickelt. Die Idee ist die Formulierung eines Optimierungsproblems auf Geschwindigkeitsebene, bei der die Lösung ausgewählt wird, die die Gelenkgeschwindigkeiten bezüglich einer Gewichtungsmatrix W minimiert: (Whitney 1969) (Siciliano et al. 2009: 124–7)

$$\dot{q}=W^{-1}\cdot J_w^T\cdot(J_w\cdot W^{-1}\cdot J_w^T)^{-1}\cdot\dot{w}_{d,eff} \qquad \textbf{( 5–5 )}$$

Durch numerische Integration entsteht ein unbegrenzter Drift, wenn man $\dot{w}_d$verwenden würde. Deshalb wird die Integration als eine Regelstrecke betrachtet, in der $\dot{w}_d$ wie folgt zu $\dot{w}_{d,eff}$ angepasst wird:

$$\dot{w}_{d,eff} = \dot{w}_d+K(w_d-w(q)) \qquad \textbf{( 5–6 )}$$

Der Faktor K muss dabei eine positiv definite Gewichtungsmatrix sein. Neben dem mathematischen Problem gibt es noch eine andere systematische Herausforderung, die zwar trivial klingen mag, jedoch in der klassischen Robotik nicht sehr häufig Anwendung findet. Dem Roboter muss es möglich sein, seine Freiheitsgrade über 360° hinaus drehen zu können. Normalweise sucht man den kürzesten Weg zu seiner Zielposition und berechnet daraus die notwendigen Gelenkwinkel und Gelenkwinkelgeschwindigkeiten. Die Eulerdarstellung ermöglicht lediglich eine Rotation von -180° bis 180°. Die Lösung dafür wird im nachfolgendem Kapitel behandelt.

## 5.3 Lösung des Problems

Für die allgemeine Lösung des inversen Problems wurde ein Code vom Lehrstuhl für Angewandte Mechanik zur Verfügung gestellt, der einen nichtredundanten Roboter mit drei Freiheitsgraden löst. Aus dieser einfachen Anwendung kann man mittels einiger Modifikationen das Problem auf das RACOON-Lab erweitern. Bei der Kompilierung in C# stellen sich weitere Schwierigkeiten heraus, die im Kapitel 6 ausführlicher behandelt werden. Aus Übersichtsgründen wird zunächst der grobe Aufbau mit allen Bestandteilen erklärt, um anschließend auf die Einzelheiten einzugehen, wenn der Code detaillierter durchgearbeitet wird.

Ein bisheriges Problem war das teilweise abrupte Verfahren des Roboters während einer Simulation. Der Computer berechnet eine Lösung ohne diese auf Basis von Randbedingungen oder Präferenzen mit anderen Lösungen zu vergleichen und die bestmögliche auszusuchen. Deshalb ist zum einen eine Nebenbedingung notwendig, zum anderen ein gesonderter Fall, in dem die direkte Kinematik vor der eigentlichen inversen Kinematik berechnet wird und der Roboter sanft zu seiner Startposition

verfährt. Die Nebenbedingung für eine eindeutige Lösung ist die in Kapitel 5.3 vorgestellte Minimierung der Gelenkgeschwindigkeiten.

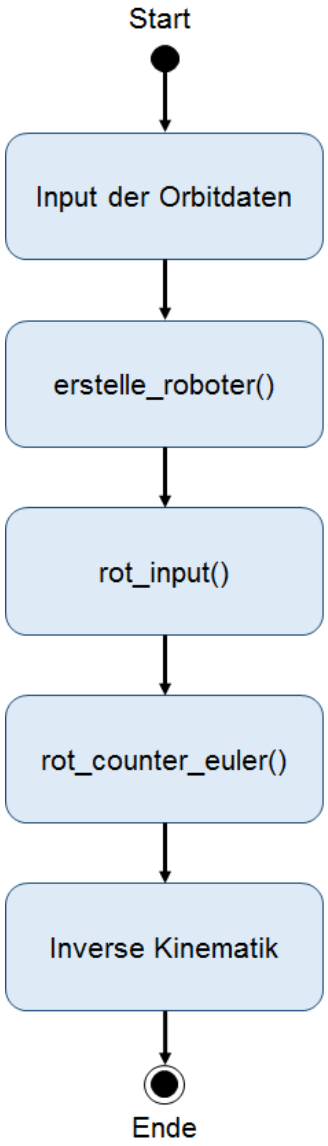



Abb. 5–3: Flowdiagramm für den Maincode

Der Code beginnt mit der Initialisierung der Orbitdaten aus dem Axismapper. Die Daten werden zuvor von ECI- auf Anlagenkoordinaten umgerechnet. Unabhängig davon wird anschließend der Roboter sowie alle Transformationsmatrizen mit Hilfe der DH-Parameter erstellt (erstelle_Roboter(), Abb. 5–3). Der Chaser kann aufgrund seiner translatorischen Einschränkung nicht jede Lage um das Target durch Translation einnehmen. Deshalb wird im nächsten Schritt der Relativvektor durch Rotationen um die z- bzw. y-Achse auf die x-Achse der Anlage gedreht (rot_input(), Abb. 5–3). Damit die relative Lage zueinander wieder richtig ist, muss die Ausrichtung von Chaser und Target ebenfalls um dieselben Beträge rotiert werden. Die Winkel werden wie folgt aus den Ortskoordinaten berechnet:

$$\gamma=\operatorname{acos}\left(\frac{x}{\sqrt{x^2+y^2}}\right) \quad \textbf{( 5–7 )}$$

$$\beta=\operatorname{acos}\left(\frac{\sqrt{x^2+y^2}}{\sqrt{x^2+y^2+z^2}}\right) \quad \textbf{( 5–8 )}$$

Abb. 5–4 zeigt die projizierten Anteile des Vektors R und den Winkel in der xy-Ebene, sowie den um γ gedrehten Vektor R‘ und den Winkel β in der xz-Ebene.

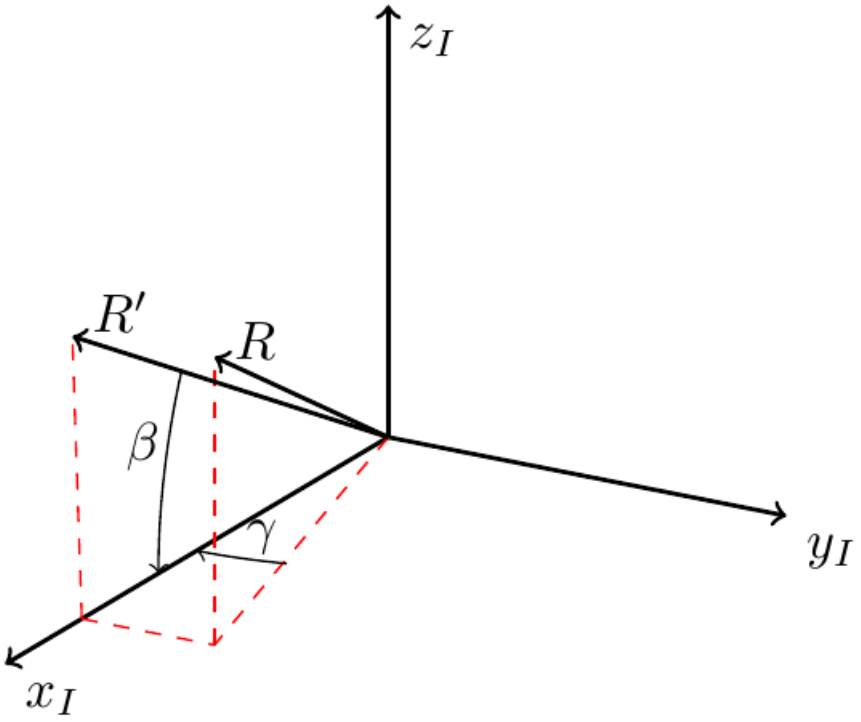



Abb. 5–4: Drehung des Ortsvektors R auf die x-Achse

Der Code ist so ausgelegt, dass er Winkel θ ∈]-∞;∞[ bearbeiten kann. Die Winkel, die mit dem Atan2 ausgerechnet werden und als Input aus der dynamischen Orbitsimulation stammen, können nur Werte zwischen -180° und 180° einnehmen.

Aus diesem Grund müssen die Winkel für den Roboter modifiziert werden, indem man einen Zähler einbaut (rot_counter_euler(), Abb. 5–3). Dieser Zähler merkt sich jedes Überschreiten des Roboters von 180° auf 181°. Da die Lösungsmenge nur zwischen -180° und 180° liegt, würde der Roboter anstatt 1° weiter zu laufen, 179° in die andere Richtung verfahren. Im umgekehrten Fall geht der Zähler um eins nach unten. Nachdem die neue Sollposition mit der aktuellen Position verglichen wurde, stellt sich das Programm mit Hilfe des Zählers darauf ein und verändert die Sollposition um 2π multipliziert mit dem Wert des Zählers (vgl. Abb. 5–5). Das ermöglicht es dem Roboter sich beliebig oft um den jeweiligen Freiheitsgrad zu rotieren. Im letzten Schritt startet der eigentliche Algorithmus der Inversen Kinematik, der nachfolgend noch genauer erläutert wird.

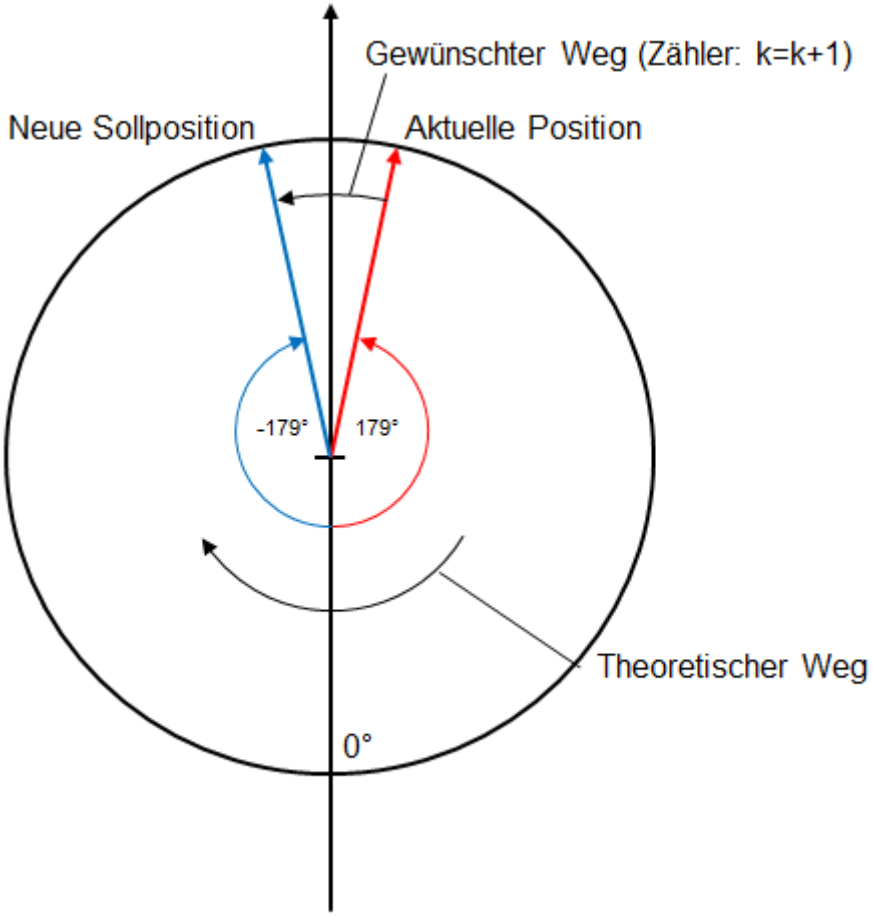



Abb. 5–5: Funktionsweise des Zählers für die Erweiterung des Winkelbereichs

Die einzelnen Schritte des Algorithmus‘ für die Inverse Kinematik werden in Abb. 5–10 dargestellt.

Der Algorithmus teilt die zurückzulegende Wegstrecke in vorher festgelegte Zeitintervalle dt ein und nähert sich der Lösung für t=T=10ms so Schritt für Schritt an. Wenn der Zielzeitpunkt noch nicht erreicht ist, beginnt er mit der Initialisierung der quadratischen Gewichtungsmatrix W, deren Diagonale die Gewichtungen für jeden Freiheitsgrad enthält. Unabhängig von der Position des Roboters müssen translatorische Freiheitsgrade gegenüber den rotatorischen stärker gewichtet werden, da der Weg für einen Meter viel größer ist, als der bei einer Rotation um 1 rad. Ausgehend von seiner aktuellen Position und den festgelegten Intervallen, in denen jeder Freiheitsgrad verfahren darf, passt das Programm die Gewichtung mittels einer „Badewannen-Funktion“ an. Die Gewichtung wird exponentiell erhöht, wenn der Roboter gegen seine physischen Grenzen fährt, damit dieser zum Stillstand kommt (vgl. Abb. 5–6). Die Funktion 5–9 beschreibt die Kurve für jede Grenzen g und h mit einem Minimum mit Wert 1:

$$W=0.5\cdot\left(\frac{-1}{(g-x)\cdot(h-x)}+1-\frac{2}{(g-h)\cdot(h-g)}\right) \qquad \textbf{( 5–9 )}$$

Die Grenzen können dabei im Bereich zwischen −2π und 2π angegeben werden, da der Algorithmus keine Beschränkung hat und nur die Ausgabe dementsprechend an

die Eulerwinkel angepasst wird. Das ermöglicht es, jeden zulässigen Bereich für ein Gelenk zu definieren.

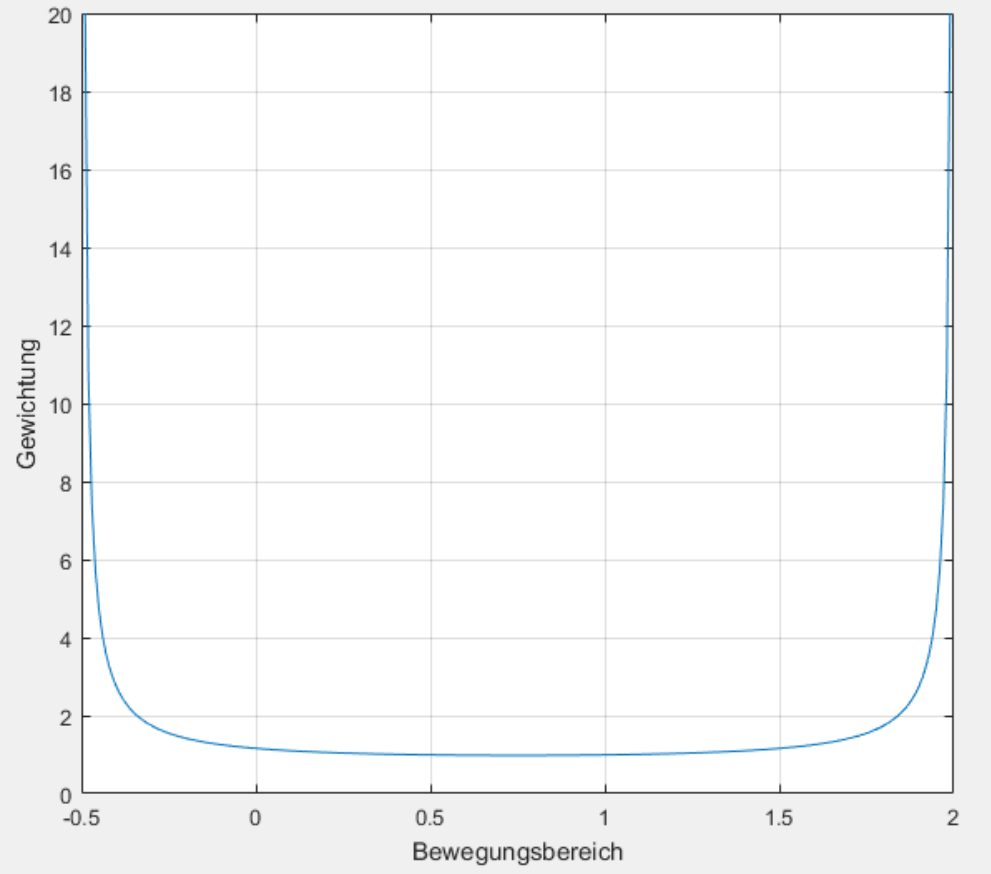



Abb. 5–6: Beispielhafte Gewichtungsfunktion für die Grenzen -0,5 und 2,0

Die Berechnung der Sollwerte von w und ẇ für den jeweiligen Zeitschritt erfolgt mittels der Ausgangs- bzw. Endposition und dem Zeitintervall. Die Theorie wurde in Kapitel 4.4 Planung der Trajektorien erläutert und das Programm unverändert vom Lehrstuhl für Angewandte Mechanik übernommen.

Für die Berechnung der aktuellen Position sowie die Ausrichtung des Roboters wird zuerst der relative Verschiebungsvektor $r_{vi}$ zum jeweiligen Vorgänger ermittelt. Dabei gehen die DH-Parameter a und d ein, die eine Verschiebung der x- bzw. z-Achse beschreiben. Beispielsweise wird hier die Verschiebung der Kamera des Chasers zu seiner z-Achse oder die translatorische Bewegung des Roboters berechnet. Aus den relativen Drehmatrizen wird für jeden Körper auch die Drehmatrix zur Basis bestimmt. Aus diesen können wiederum über den Atan2 die Eulerwinkel berechnet werden, welche die Arbeitsraumkoordinaten des Roboters vervollständigen. Dieses Programm wurde teilweise vom AMM übernommen und für das RACOON-Lab angepasst.

Nachdem nun die Sollposition und die aktuelle Position des Roboters bekannt sind, muss geprüft werden, ob sich die Gelenke einer Singularität nähern würden. Die Singularität bei der Berechnung der Eulerwinkel liegt bei $\gamma=\pm 90°$ (Rotation um die y-Achse) (Craig 2014: 43). Für diese Fälle wird mittels „Singularity Avoidance“ das gesamte Referenzsystem um 90° gedreht, die Lösung generiert und anschließend

wieder um 90° zurückgedreht. Das Drehen des Inertialsystems ist eine elegante Methode, um Singularitäten aus dem Weg zu gehen. (Singla P., Mortari D. and Junkins J. L. 2005: 10)

Abb. 5–7 zeigt beispielsweise eine Rotation der y-Achse in Richtung 90°. Falls die aktuelle Position (rot) sich der Singularität nähert, wird diese um -90° verdreht, um dieser aus dem Weg zu gehen.

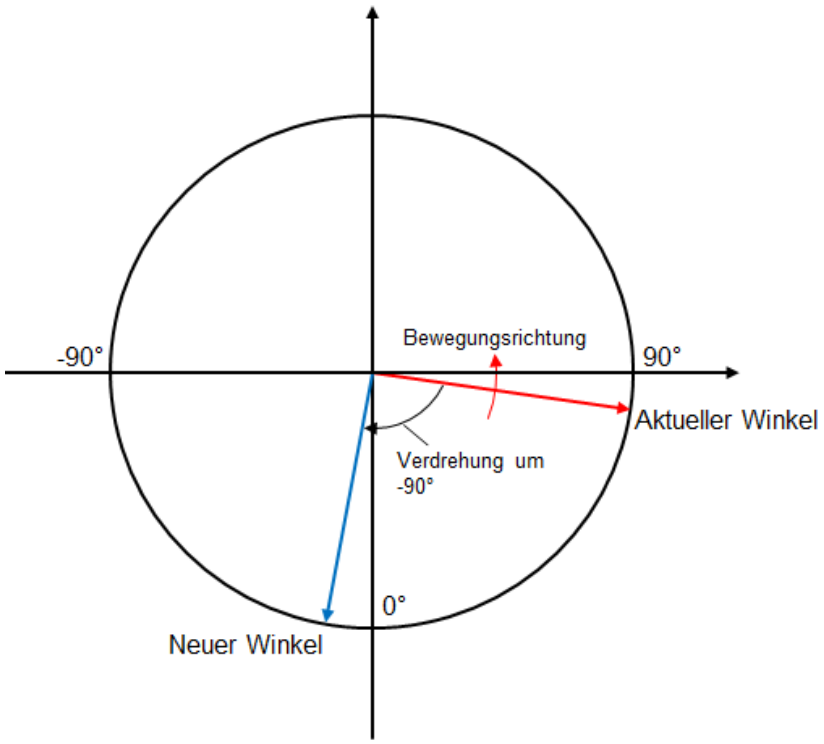



Abb. 5–7: Veranschaulichung der "Singularity Avoidance"

Um die Notwendigkeit der „Singularity Avoidance“ zu verdeutlichen, zeigen Abb. 5–8 und Abb. 5–9 die Ergebnisse des Algorithmus‘ mit und ohne dieser für eine Rotation um die y-Achse um insgesamt 90°. Ab ca. 63° beginnen die Werte zu divergieren und unrealistische Zahlen hervorzubringen. Die Lösungen befinden sich nur innerhalb ±π, da der Algorithmus die Ausgabe in Eulerwinkel transformiert und keine außerhalb dieses Bereiches zulässt. Die Achse „Target y“ wurde für diese Berechnung mit dem Faktor 5 behaftet, damit die Kurven für diese und „Target C-Sled“ nicht übereinander fallen.

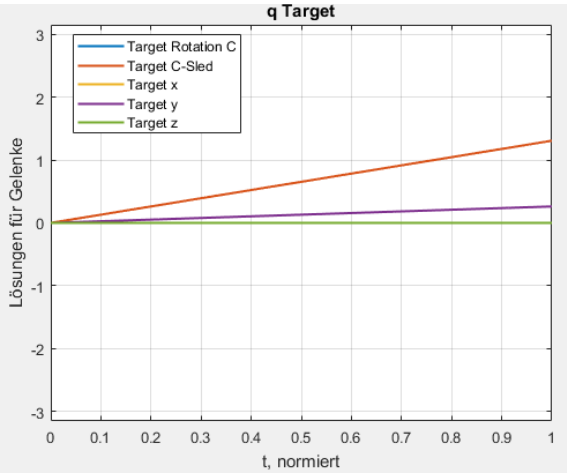



Abb. 5–8: 90° Rotation des Targets um die y-Achse mit Sinularity Avoidance

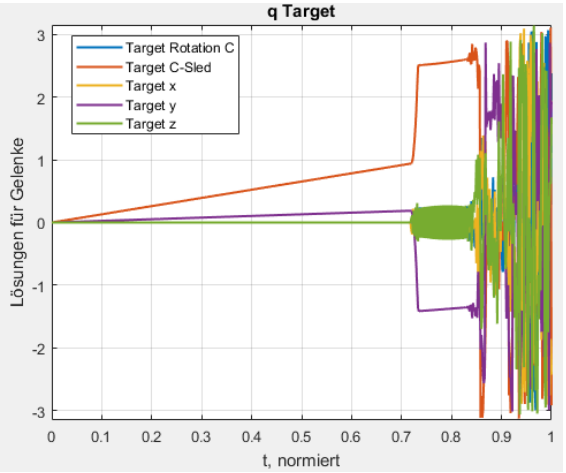



Abb. 5–9: 90° Rotation des Targets um die y-Achse ohne Sinularity Avoidance

In der Robotik unterscheidet man zwischen einer rotatorischen und einer translatorischen Jakobimatrix bzw. zwischen Dreh- und Schubgelenk. Die Berechnung für die rotatorische Jakobimatrix wurde aus dem Code vom AMM übernommen und um die Berechnung für translatorische Jakobimatrix erweitert sowie an die Parametrisierung angepasst (aus Gl. 4−23):

$$J_{T,rel,i}=A_{i,i-1}\cdot(\tilde{r}_{i-1,i}\cdot J_{R,i-1})+A_{i,i-1}\cdot A_x^T(\boldsymbol{\alpha_i})\cdot e_z \qquad \textbf{( 5–10 )}$$

Es wird nicht mehr um den Winkel α des Vorgängers i-1, sondern um den aktuellen Winkel $\alpha_i$ gedreht.

Die zweite Änderung ist, dass $J_{R,rel,i,[m,n]}$ für m=3 und n=q zu −1 gesetzt wird. Normalerweise wird $J_{R,rel,i,[m,n]}=1$ gesetzt, jedoch divergiert hier der Algorithmus aus unbekannten Gründen. Die einzige Änderung ergibt sich, dass die Gelenkwinkel bei der Ausgabe negativ gesetzt werden müssen. Deshalb werden die Systemgrenzen auch automatisch im Programm mit −1 multipliziert, damit diese wieder mit den Vorzeichen der Gelenkwinkel konsistent sind. Für die Eingabe, bei einer möglichen Anpassung der Grenzen, ändert sich somit nichts und kann, wie in Kapitel 6.1 beschrieben, durchgeführt werden.

Für die neun Arbeitsraumkoordinaten, die drei Raumrichtungen und die jeweils drei Eulerwinkel, kann man sich aus den Jakobimatrizen eine einzige zusammenbauen, da diese nur angibt, welcher Freiheitsgrad auf welche Arbeitsraumkoordinate Einfluss hat. Das bedeutet für die Translation verwendet man die translatorische bzw. für die Rotation eine rotatorische Jakobimatrix. Im Speziellen die rotatorische Jakobimatrix für den Chaser und zusätzlich die für das Target. Im letzten Schritt müssen die Jakobimatrizen noch vom Relativsystem i zur Basis 0 transformiert werden, um die endgültige Jakobimatrix zu erhalten:

$$J=\begin{pmatrix} A_{i0}^T J_T \\ A_{i0}^T J_{R,Chaser} \\ A_{i0}^T J_{R,Target} \end{pmatrix} \qquad \textbf{( 5–11 )}$$

Im Main-Programm (vgl. Abb. 5–3, rot_counter_euler()) wurde bereits einmal die neue Soll-Position mit der alten Soll-Position verglichen, jedoch ohne die Ist-Position mit einzubeziehen. Das ist insofern wichtig, da sich der Roboter aufgrund von numerischen Fehler nicht exakt auf seiner Sollposition befindet. Beispielsweise, wenn der Zähler auf eins gesetzt wird um einen Winkel anzupassen, der Gelenkwinkel des Roboters aber noch nicht angepasst werden muss, würde man fälschlicherweise den Gelenkwinkel um 2π erhöhen. In anderen Worten, durch numerische Fehler und in speziellen Fällen, eben nahe den Grenzen -180° und 180°, müssen nicht alle Soll- und Ist-Werte angepasst werden.

Der nächste Schritt ist logischerweise die Veränderung der Arbeitsraumkoordinaten w, damit der Algorithmus mit diesen arbeiten kann. Da nun die Soll-Position und die Position des Roboters die für den Algorithmus korrekten Werte besitzen, wird die Differenz bzw. der Fehler dw berechnet. Aus dieser Differenz bestimmt man die effektive Soll-Geschwindigkeit $\dot{w}_{d,eff}$ (vgl. Gleichung 5–6), die benötigt wird um die Gelenkgeschwindigkeiten zu berechnen (vgl. Gleichung 5–5). Mittels numerischer Integration erhält man die gesuchten Gelenkwinkel:

$$q_{f+1}=q_f+\dot{q}_f \cdot dt \qquad \textbf{( 5–12 )}$$

Falls im Schritt „Singularity Avoidance“ eine Rotation stattgefunden hat, muss diese wieder korrigiert werden, um die tatsächlichen Winkel zu erhalten. Dabei fließen die Gewichtungen der einzelnen Gelenke mit ein, da der Roboter redundant ist und mehrere Gelenke zu einer Arbeitsraumkoordinate beitragen.

Der Algorithmus wird solange durchlaufen bis das Zeitinkrement T=10ms erreicht ist. In Kapitel 6 werden zudem die Auswirkungen auf die Wahl des Zeitintervalls mit der Schnelligkeit der Berechnung und der Genauigkeit analysiert.

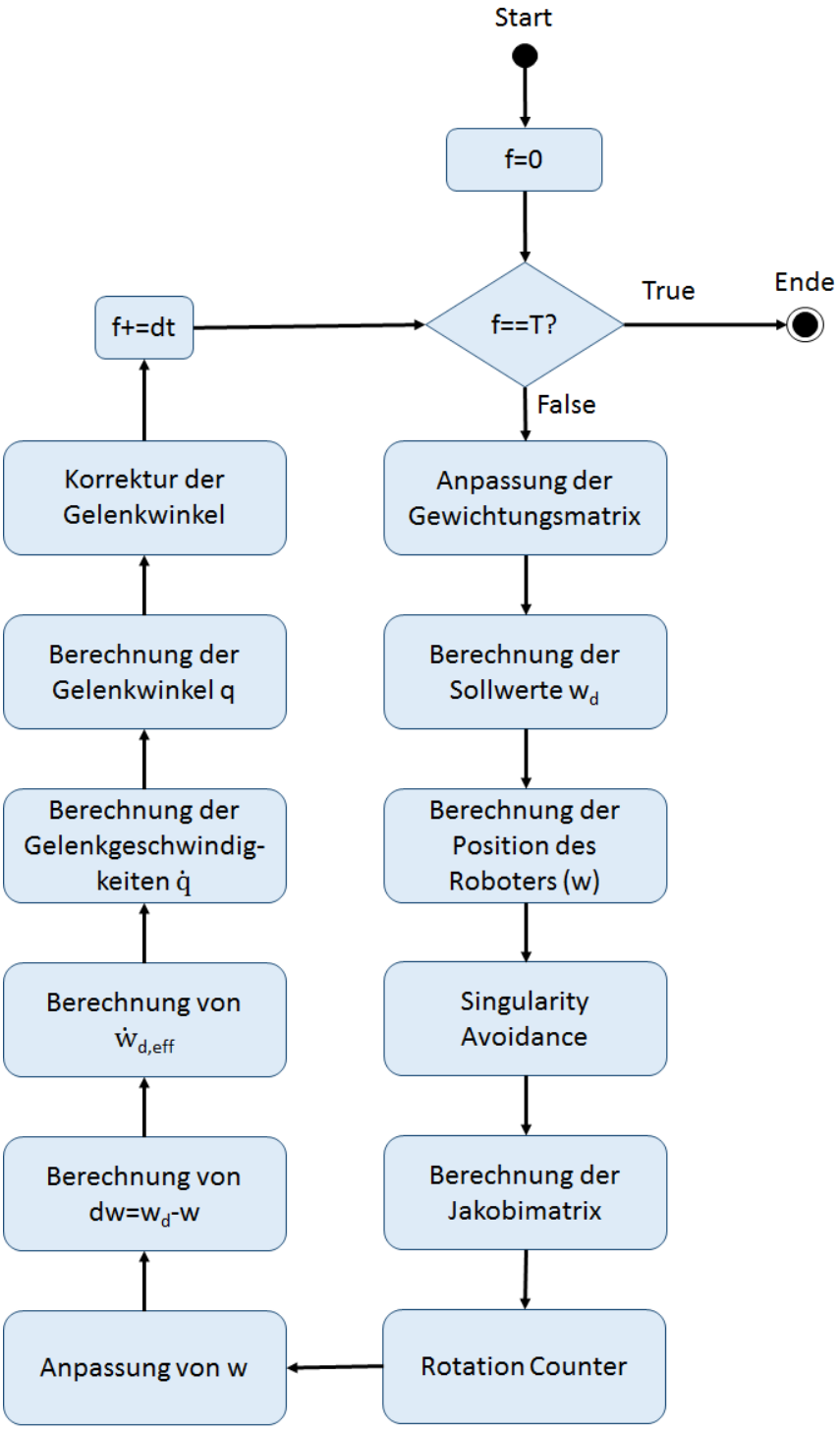



Abb. 5–10: Algorithmus zur Berechnung der Inversen Kinematik

# 6 Implementierung

Für die Implementierung wurde MATLAB verwendet, da zum einen der Vorteil der schnellen Überprüfbarkeit der Ergebnisse durch Plots besteht, zum anderen gibt es in MATLAB diverse Compiler, die den Code in C# umwandeln können. Wie bereits im vorherigen Kapitel erwähnt, wurde für diese Arbeit ein Code für die Berechnung eines nicht-redundanten Roboters mit drei rotatorischen Freiheitsgraden vom Lehrstuhl für Angewandte Mechanik zur Verfügung gestellt. Die verantwortlichen Mitarbeiter Felix Sygulla und Daniel Wahrmann baten mir ihr Hilfe für das Einarbeiten in den Code an und standen jederzeit für Fragen zum Code oder zum Thema Inverse Kinematik zur Verfügung. Deshalb möchte ich mich an dieser Stelle nochmal für die Unterstützung bedanken.

## 6.1 Implementierung in MATLAB

Generell sind alle Teile durch Funktionen aufgebaut. Ein Hauptprogramm ruft andere Unterprogramme auf und übergibt die benötigen Werte. Um diese Werte kompakt zu halten wird für den Roboter eine Klasse erstellt, die alle Variablen und Matrizen beinhaltet. Diese Klasse ist mit Feldern strukturiert, um nicht mit bloßen Zahlen arbeiten zu müssen. Die Felder werden nach den einzelnen Variablen und Transformationsmatrizen benannt. Das ermöglicht es die Korrektheit der implementierten Formeln für die Berechnungen schnell zu überprüfen und Fehlerquellen durch Zahlendreher zu minimieren. Tab. 6–1 zeigt alle Bestandteile der Klasse „Roboter“ auf. Neben den Variablen und Matrizen werden auch die Zähler mit abgespeichert. Die Reihenfolge entstand durch sukzessives Hinzufügen neuer Zähler und Werte, die bei der Entwicklung des kompletten Codes nach und nach gebraucht wurden.
Das Mainprogramm bekommt als Eingabe die auf die Anlage umgerechneten Ortsvektoren von „TransformChaser2Facility.Translation“ und „TransformTarget2Facility.Translation“ sowie Eulerwinkel von „(*ChaserBaseData).RotationBase2Eci.GetEulerZYX“ und „(*TargetBaseData).RotationBase2Eci.GetEulerZYX“ aus der Orbitsimulation in Visual Studio. Diese bilden die Arbeitsraumkoordinaten, die für die weitere Berechnung benötigt werden. Die Ausgabe besteht aus der Klasse Roboter mit den in Tab. 6–1 aufgelisteten Bestandteilen.

Tab. 6–1: Auflistung der Bestandteile der Klasse Roboter

| Zeilenindex | Bestandteile der Klasse Roboter | Größe (Zeilen x Spalten) |
|---|---|---|
| **1** | Anzahl der Koordinatensysteme | 1 x 1 |
| **2** | Matrix mit Variablen, DH-Parameter und Transformationsmatrizen | 30 x 16 |
| **3** | Anzahl der Freiheitsgrade | 1 x 1 |
| **4** | Arbeitsraumkoordinaten | 9 x 1 |
| **5** | Gelenkkoordinaten | 11 x 1 |
| **6** | Gelenkgeschwindigkeiten | 11 x 1 |
| **7** | Gelenkbeschleunigung | 11 x 1 |
| **8** | Aktueller Zeitschritt | 1 x 1 |
| **9** | Jakobimatrix | 9 x 11 |
| **10** | Zähler für ±180° Überschreitungen | 6 x 2 |
| **11** | Sollposition, alte Sollposition, alte Istposition | 9 x 3 |
| **12** | Zähler für Singularity Avoidance | 6 x 1 |
| **13** | Physische Grenzen der Anlage | 11 x 2 |
| **14** | Startposition des Roboters | 9 x 1 |
| **15** | Zähler für die Rotation der Inputdaten | 2 x 1 |
| **16** | Zeitdauer für die Berechnung | 1 x 1 |
| **17** | Schalter für Direkte Kinematik | 1 x 1 |

Der Code muss insofern erweitert werden, da der Roboter mit drei Freiheitsgraden keine Schubgelenke, sondern nur Drehgelenke besitzt und die Berechnung der Jakobimatrizen deshalb angepasst werden muss. In Kapitel 5.2 wurde bereits das mathematische Problem für einen redundanten Roboter und den damit verbundenen Auswirkungen diskutiert. Die nicht-quadratische Jakobimatrix führt zu einer pseudoinversen Jacobimatrix, die mit Hilfe einer Gewichtungsmatrix gebildet wird. Diese Gewichtungsmatrix ist zudem hilfreich, um die Bewegungsfähigkeit des Roboters einzuschränken und diesen an seine physischen Grenzen anzupassen. Folgende Programme sind im Algorithmus enthalten:

- Scene (=Mainprogramm)
- IK (=Inverse Kinematik)
- Erstelle_roboter
- Berechne_dk_positionen
- Berechne_dk_jacobis
- Ax
- Az
- Rot_input

- Rot_counter_euler
- Rotationcounter
- Singularity Avoidance
- Linkcorrection
- Sollwerte_w
- Tilde
- init

Die Grenzen für die Freiheitsgrade des Roboters können im Programm „erstelle_roboter()“ unter dem Block „%% Grenzen“ am Anfang des Codes verändert werden. Die Werte in Spalte eins bilden die Grenzen bei einer positiven Drehung und die Werte in Spalte zwei die Grenzen bei einer negativen Drehung ausgehend von der Ruheposition. Das Programm „init“ initialisiert den Roboter und deklariert diesen als Variable.

## 6.2 Kompilierung und Einbindung in Visual Studio

Nachdem Eingabe, Ausgabe und die Struktur des Algorithmus feststehen, muss der Code noch kompiliert werden, um ihn in Visual Studio verwenden zu können. Dafür wurde zuerst der in MATLAB integrierte Coder verwendet, der sich als weniger geeignet herausgestellt hat. Damit der Code in dieser App kompiliert werden kann, müssen die Dimensionen alle Variablen und Matrizen im Vorhinein bekannt sein und mit Werten initialisiert werden. Ebenso muss bei jeder Verwendung einer Variablen im Code die Dimension angegeben werden. Des Weiteren ergeben sich Probleme bei der Verwendung von Feldern zur Strukturierung der Daten, was den Vorteil der Übersichtlichkeit durch eine eindeutige Benennung verschwinden lässt. Damit man die Berechnungen nachvollziehen kann, stellt Tab. 6–2 die Verknüpfung der Variablen und Matrizen mit den Spaltenindizes dar (für die Matrix in der Tab. 6–1, Zeilenindex 2). In anderen Worten, die Klasse Roboter ist nun ein Vektor, dessen Einträge wiederum Vektoren und Matrizen sind, aus dem man mittels Zeilen- und Spaltenindizierung die gewünschten Werte bekommt. Der MATLAB Coder kann mit normalen Arrays nicht arbeiten und benötigt deshalb, für diese Art der Struktur, Zellen, was zusätzlich mit einem großen Aufwand an Umprogrammierung verbunden ist. Außerdem müssen nach der Initialisierung alle Variablen als solche mit „coder.varsize(‘Name des Vektors/der Matrix‘); gekennzeichnet werden.

Die ersten Versuche den Code zu kompilieren waren nicht erfolgreich und nach einiger Recherche stellte sich heraus, dass die App in der Version MATLAB R2017a unvorhergesehen abstürzt und man die Version MATLAB R2016b verwenden muss. Zusätzlich muss auf dem Computer ein eigener Compiler für Visual Studio installiert sein. Da diese App einige Komplikationen mit sich bringt und noch keine erfolgreiche Kompilierung entstand, wurde für die nächsten Versuche der in MATLAB integrierte „Library Compiler“ verwendet. Die Kompilierung mit dem Library Compiler erfolgt problemlos mit dem Compile-Typ „.NET Assembly“, der die für Visual Studio benötigte dll-Datei hervorbringt.

Es ist nicht nachvollziehbar, welche Änderungen, die für die Kompilierung mit dem MATLAB Coder gemacht wurden, ebenfalls für die Kompilierung mit dem Library Compiler nötig sind. Ein weiterer Vorteil ist die sehr viel schnellere Kompilierung

gegenüber dem Coder, da dieser unter anderem zuerst nach Run-Time Fehler sucht und danach zu kompilieren beginnt.

Tab. 6–2: Verknüpfung der Spaltenindizierung mit den Bezeichnungen

| Spaltenindex | Bezeichnung | Spaltenindex | Bezeichnung |
|---|---|---|---|
| **1** | Name | **9** | $A_{iv}$ |
| **2** | Vorgänger | **10** | $A_{i0}$ |
| **3** | Freiheitsgrad | **11** | $r_i$ |
| **4** | α | **12** | $r_0$ |
| **5** | a | **13** | $D_i$ |
| **6** | θ | **14** | $D_0$ |
| **7** | d | **15** | $J_{R,i}$ |
| **8** | $r_{vi}$ | **16** | $J_{T,i}$ |

Für die Kompilierung ist anzumerken, dass für den Namen der Datei „Axismapper_IK“ gewählt und die Klasse dem entsprechend mit „Axismapper_IK_class“ benannt wurde. Diese sollten bei einer Änderung im MATLAB-Code beibehalten werden, damit man den Code in Visual Studio nicht anpassen muss, sondern die neue Version sofort verwenden kann.

Nach erfolgreicher Kompilierung in eine dll-Datei wird diese mittels „Projektmappen-Explorer -> Projektmappe „axismapper“ -> FacilityAxisMapper -> Verweise -> Verweis hinzufügen“ referenziert. Das Programm kann auf die dll-Datei zugreifen, wenn diese im Code durch „using ‚Name der Datei‘;“ in den Axismapper eingebunden wird. Die Erweiterung des Codes für den Axismapper erfolgt in

- Public unsafe override void CallbackFunctionProcessStates()

unmittelbar bevor die Funktion “UpdateAxisPositionVelocity“ aufgerufen wird.

Da die Methode dynamisch ist, muss eine lokale Variable erstellt werden. Das wird beispielhaft für eine RACOON_Axismapper.dll, deren Klasse mit RACOON benannt wird und der lokalen Variablen LV gezeigt:

- RACOON_Axismapper.RACOON LV=new RACOON_Axismapper.RACOON();

Der Name der Datei und der Klasse kann vor dem Kompilieren festgelegt werden.

MATLAB benötigt als Eingabeformat das MWArray. Da die Vektoren bzw. Matrizen im Vector3D- oder Matrix4D-Format sind, müssen diese entweder umformatiert werden oder als „double“ übergeben werden. Das bedeutet, zuerst werden alle benötigten Variablen als „double“ initialisiert und anschließend die entsprechenden Werte zugewiesen. Das Main-Programm „Scene“ der Inversen Kinematik kann dann über die lokale Variable und den Eingabewerten aufgerufen werden:

- MWArray roboter = LV.Scene (Vektorkomponenten der Ortsvektoren und der Eulerwinkel, Zeitintervall dt);

Die Ausgabe wird mit folgendem Befehl richtig formatiert und die Zeilen i des Ausgabevektors „roboter“ mit Roboter-i nummeriert:

- double[,] Roboter-i = (double[,]) ((MWNumericArray) roboter[i]).ToArray (MWArrayComponent.Real);

Die berechneten Werte für die Gelenke werden anschließend “ChaserPositioner.Links[i].q” und “TargetPositioner.Links[i].q” zugewiesen, damit die Positionen aktualisiert werden können.

Für die Simulation ist es wichtig, dass im Control Terminal für Target und Chaser „in Frame“ und „Position“ zu „ECI“ und „Base“ gesetzt werden, da der Algorithmus dieses Referenzsystem zur Verarbeitung der Daten braucht.

# 7 Verifikation

Die in dieser Arbeit entstandene Beschreibung der Kinematik des RACOON-Lab wird zunächst in MATLAB verifiziert, da die Visualisierung der Ergebnisse sehr einfach und schnell funktioniert. Anschließend folgt eine Verifikation in Visual Studio mit der Orbitsimulation. Ein weiterer wichtiger Punkt ist die Analyse der Laufzeit und der Rechengenauigkeit, die sowohl in MATLAB als auch in Visual Studio ausgewertet und verglichen werden.

## 7.1 Verifikation in MATLAB

Für die Verifikation in MATLAB werden zwei unterschiedliche Szenarien betrachtet, um die Funktionsfähigkeit des Algorithmus‘ weitreichend zu testen. Ein häufiges Manöver stellt der Umflug des Chasers um das Target dar, das sich entweder auch selbst bewegen kann oder fast ohne jegliche Eigenrotation im Raum schwebt. Die Eingabedaten werden hierfür durch Vektoren in MATLAB erzeugt und an den Algorithmus übergeben, der dann nötigen Gelenkwinkel berechnet. Untersucht wird ebenso die Rechengenauigkeit, sowie die dafür benötigte Zeit. Die Rechengenauigkeit lässt sich mit der Norm der Differenz aus Soll- und Ist-Position und die Rechenzeit mittels tic/toc-Befehl im MATLAB-Code bestimmen.

### 7.1.1 Scenario 1

Für das erste Szenario soll der Chaser einen elliptischen Umflug um das Target mit den Halbachsen a=5m und b=4m vollziehen. Dabei soll der Chaser stets auf das Target ausgerichtet sein und das Target in einer beliebigen Anfangsposition verharren. Das bedeutet die x- und y-Koordinate des Chasers, sowie dessen Ausrichtung bzw. die Rotation um die z-Achse werden durch den Eingabevektor bestimmt. Um die Situation realistisch zu gestalten, wird ein Flug mit geringer Winkelgeschwindigkeit des Chasers um das Target von 3 °/s simuliert. Für eine Aktualisierung der Lage in 10 ms Schritten hat das eine Winkeländerung von 0,03 °/Zeitintervall zur Folge. Zur Vereinfachung des Eingabedatensatzes wird für dieses Beispiel die Geschwindigkeit des Chasers als konstant angenommen, obwohl diese bei einer elliptischen Bahn nicht konstant sein würde. Kurven, die nicht auf den Grafen zu finden sind, überschneiden sich auf der x-Achse, da diese konstant den Wert null besitzen.

Die Randbedingungen für die drei Berechnungen mit verändertem Zeitintervall sind aus Tab. 7–1 zu entnehmen:

Tab. 7–1 Randbedingungen für drei Berechnungen in MATLAB

| Parameter | Wert |
|---|---|
| **T** | 10ms |
| **dt** | 1ms/2.5ms/5ms |
| **ω** | 0.03°/10ms |
| **Target C-Sled Gewichtung** | 5 |
| **Prozessor** | Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz |
| **Punkte gesamt** | 12001 |

und zeigen die Sollbahn des Chasers, beginnend beim Punkt (5,0) und sich in der angedeuteten Pfeilrichtung weiterbewegend, sowie die Sollwerte für die Rotation des Targets, wenn der Chaser anstatt sich um das Target zu bewegen, an Ort und Stelle verharrt und sich das Target in die entgegengesetzte Richtung dreht. Im Punkt (0,0) befindet sich das Target. In den folgenden Abbildungen sind die Koordinaten x, y und z in [m] und alle Winkel in [rad] aufgetragen.

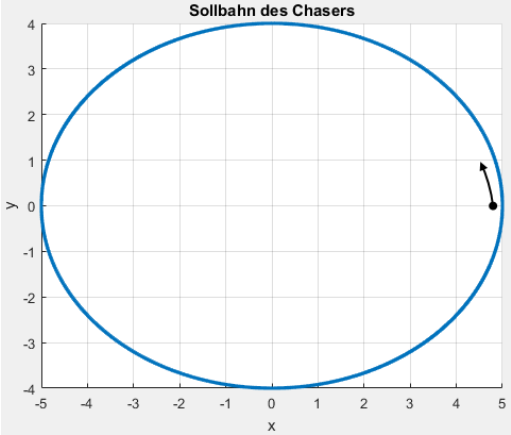



Abb. 7–1: Elliptische Sollbahn des Chasers

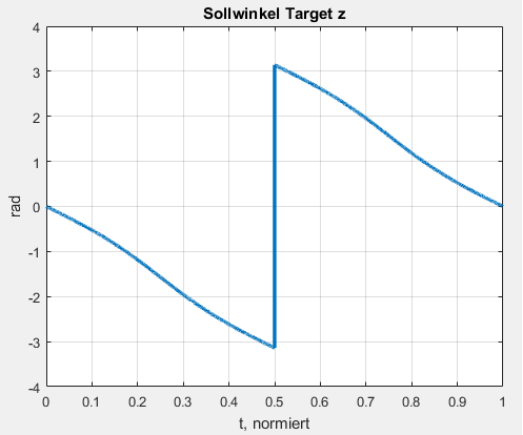



Abb. 7–2: Sollwerte für die Rotation des Targets um die z-Achse

Bei allen Zeitintervallen erfolgt die Berechnung außerhalb der Bereiche ±90° und ±180° problemlos und erzielt gute Werte. Abb. 7–3 bis Abb. 7–14 zeigen die Lösungen für die Gelenkkoordinaten und die dazugehörigen Arbeitsraumkoordinaten. Die x-Koordinate des Chasers oszilliert zwischen +6 und +7, da der Kopf des Chasers mit einem Abstand von -2 zu seiner z-Rotationsachse mitberücksichtigt wurde. Aufgrund der positiven Richtung der x-Achse vom Target zum Chaser muss sich dieser etwas weiter entfernen, um den richtigen Abstand aufrecht zu erhalten.

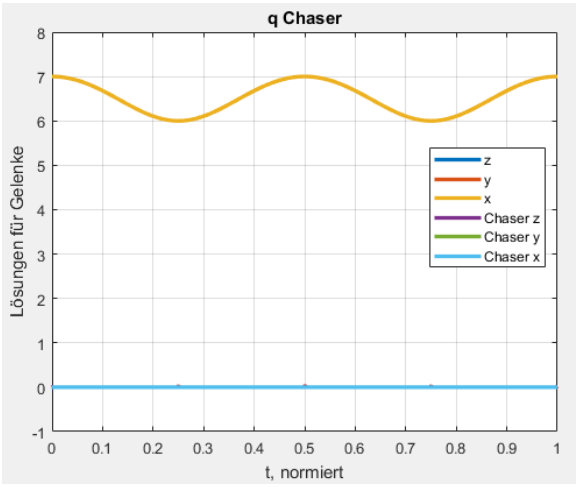



Abb. 7–3: Gelenkkoordinaten q für den Chaser mit dt=1ms

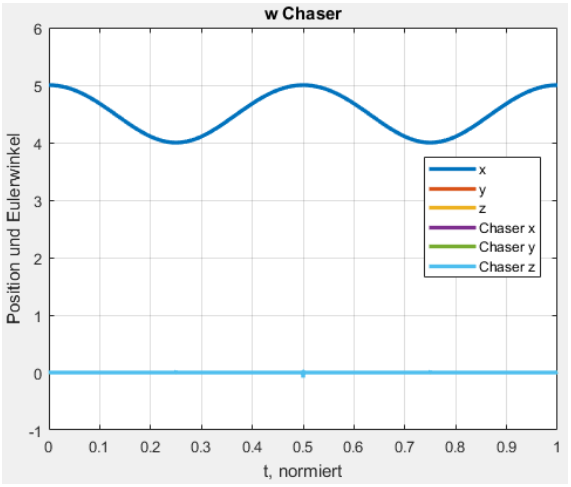



Abb. 7–4: Arbeitsraumkoordinaten w für den Chaser mit dt=1ms

In Abb. 7–3 und Abb. 7–4 bewegt sich der Chaser nur in x-Richtung, alle anderen Koordinaten sind null, dennoch sind bei t=0.25, t=0.5 und t=0.75 kleine Abweichungen zu erkennen, die in den folgenden Abbildungen noch deutlicher werden. Diese entsprechen eben jenen Stellen von ±90° und ±180°.

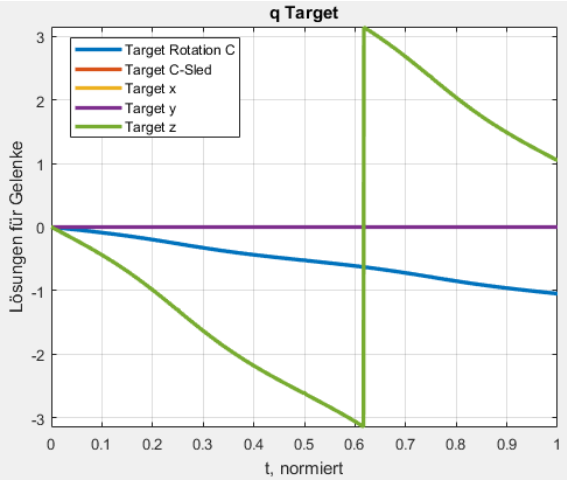



Abb. 7–5: Gelenkkoordinaten q für das Target mit dt=1ms

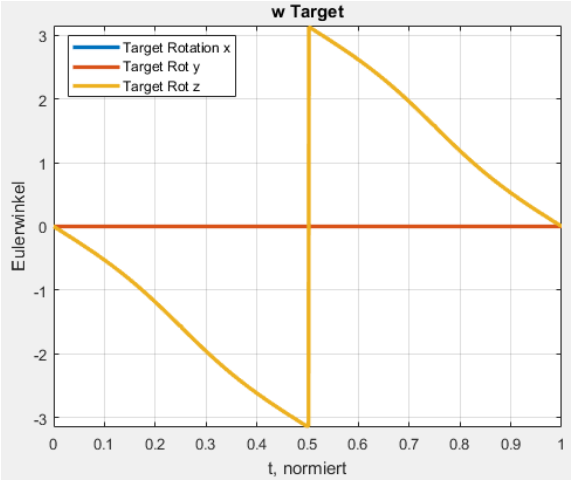



Abb. 7–6: Arbeitsraumkoordinaten w für das Target mit dt=1ms

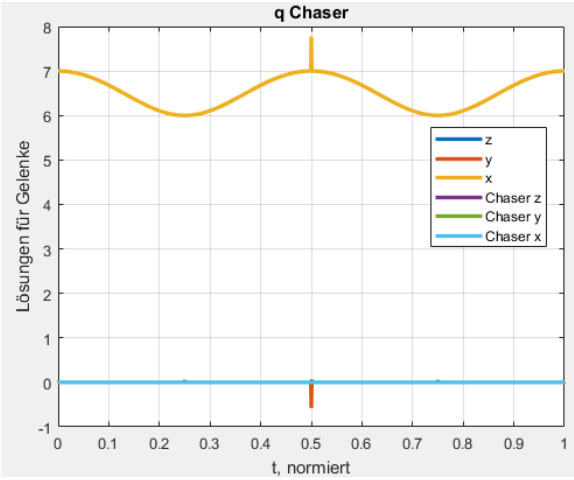



Abb. 7–7: Gelenkkoordinaten q für den Chaser mit dt=2.5ms

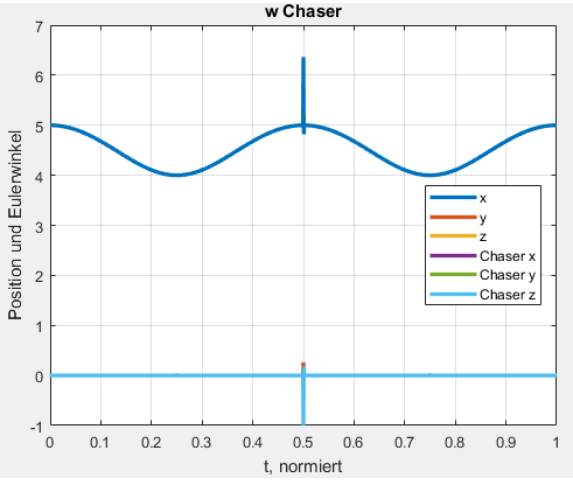



Abb. 7–8: Arbeitsraumkoordinaten w für den Chaser mit dt=2.5ms

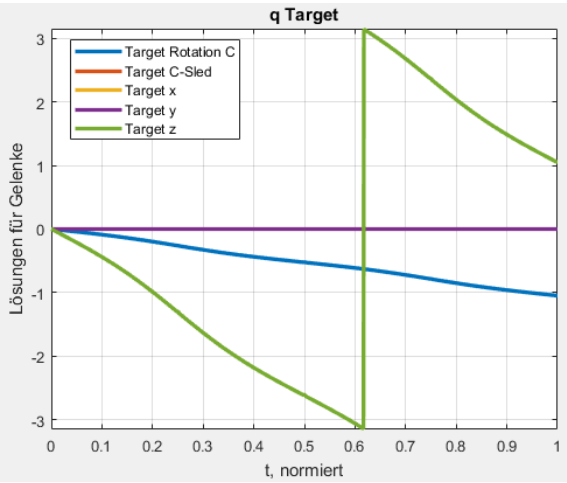



Abb. 7–9: Gelenkkoordinaten q für das Target mit dt=2.5ms

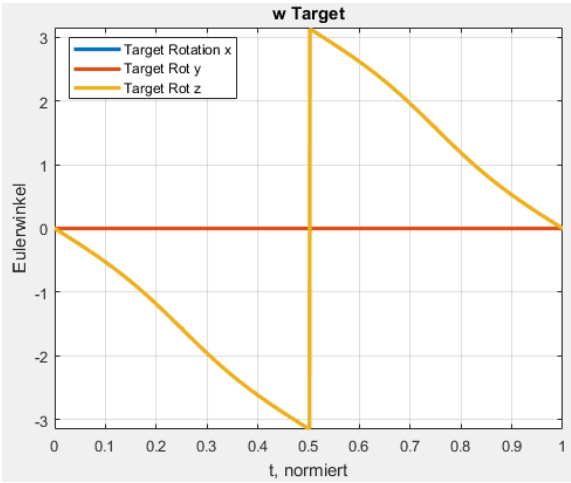



Abb. 7–10: Arbeitsraumkoordinaten w für das Target mit dt=2.5ms

Für das Zeitintervall dt=2.5ms sind in den dazugehörigen Abb. 7–7 und Abb. 7–8 an den vorher genannten kritischen Stellen deutlichere Fehler zu erkennen.

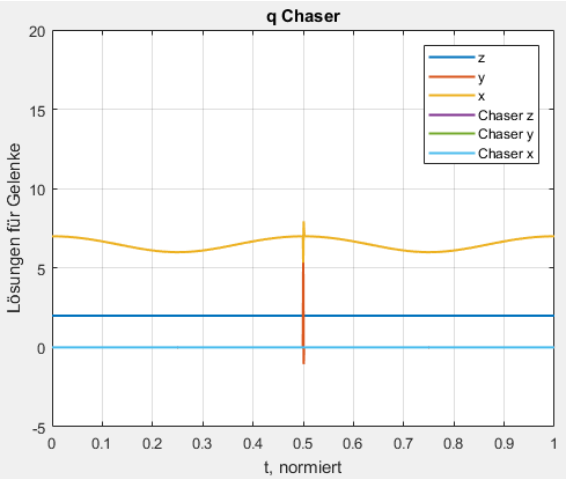



Abb. 7–11: Gelenkkoordinaten q für den Chaser mit dt=5ms

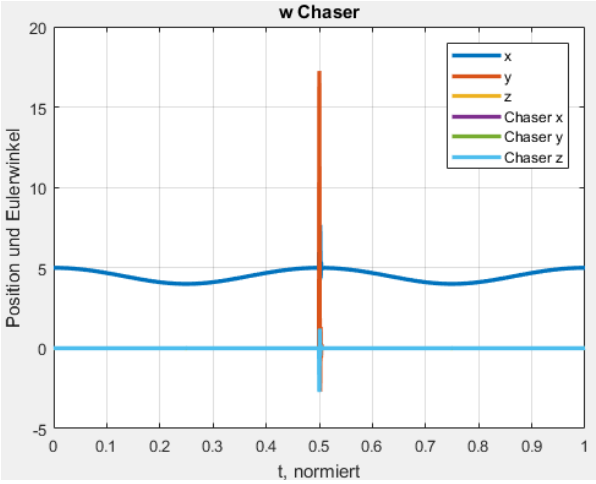



Abb. 7–12: Arbeitsraumkoordinaten w für den Chaser mit dt=1ms

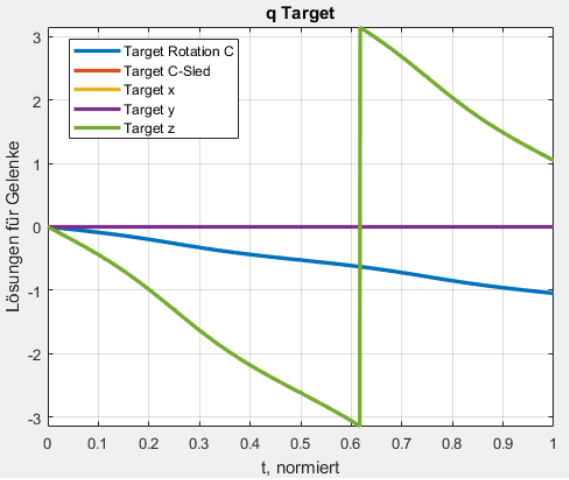



Abb. 7–13: Gelenkkoordinaten q für das Target mit dt=5ms

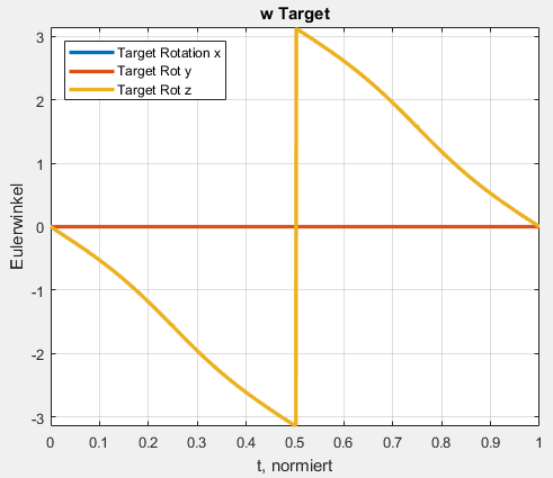



Abb. 7–14: Arbeitsraumkoordinaten w für das Target mit dt=5ms

In den Abb. 7–11 bis Abb. 7–14 steigt der maximale Fehler für das Zeitintervall dt=5ms an den kritischen Stellen noch stärker an

Für die Vergleichbarkeit der Berechnungen werden der maximale Fehler, der durchschnittliche Fehler und die durchschnittliche Berechnungsdauer ausgewertet (siehe Tab. 7–2).

Tab. 7–2: Ausgewertete Daten für die Berechnungen mit dt=1ms/2.5ms/5ms

| dt | 1ms | 2.5ms | 5ms |
|---|---|---|---|
| **Max. Fehler** | 0.098 | 1.735 | 17.911 |
| **Ø Fehler** | 1.8e-5 | 2.6e-4 | 0.0025 |
| **Ø Berechnungsdauer** | 0.0271 s | 0.0117 s | 0.007 s |

Des Weiteren ist zu beachten, dass die Aktualisierung der Lage nicht alle 10ms, sondern erst nach jedem Berechnungsschritt stattfindet. Das bedeutet, bei einer Aktualisierungsrate von 10ms und einer Berechnungsdauer >10ms, würde der Roboter mit jeder Berechnung weiter hinterherhinken bzw. der Fehler immer größer werden. Das ist jedoch nur ein Problem in MATLAB, da im Axismapper immer die aktuellste Sollposition übergeben wird und kein Vektor mit Sollwerten in 10ms-Schritten.

### 7.1.2 Scenario 2

Die nächste Simulation soll gekoppelte Rotationen um mehrere Achsen und realistische Grenzen für den Roboter beinhalten. Der Chaser fliegt eine halbe elliptische Bahn um das Target, dessen z-Koordinate sinusförmig schwingt. Das Target dreht sich zusätzlich um 90° die x-Achse. Wie im Test davor, ist der Chaser immer auf das Target ausgerichtet. Die verwendeten Systemgrenzen sind aus Tab. 5–6 zu entnehmen. Folgende Randbedingungen sind für Scenario 2 gewählt worden:

Tab. 7–3: Randbedingungen für Scenario 2

| Parameter | Wert |
|---|---|
| **T** | 10ms |
| **dt** | 2.5ms |
| **ω** | 0.03°/10ms |
| **Target C-Sled Gewichtung** | 2 |
| **Prozessor** | Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz |
| **Punkte gesamt** | 6001 |
| **Ø benötigte Rechenzeit** | 16ms |

Abb. 7–15 zeigt die elliptische Sollbahn des Chasers in Blau und das Target, welches mit einem roten „+“ gekennzeichnet ist. Der schwarze Pfeil gibt die Bewegungsrichtung ausgehend von der Startposition an. Der Chaser befindet sich vor dem Target und aus Gründen der Übersicht wurde der Plot um 180° gedreht, damit die halbe Ellipse besser zu erkennen ist. In Abb. 7–16 sind die Sollwinkel für das Target, die Rotation von 90° um die x-Achse sowie die Rotation von −180° um die z-Achse, zu sehen.

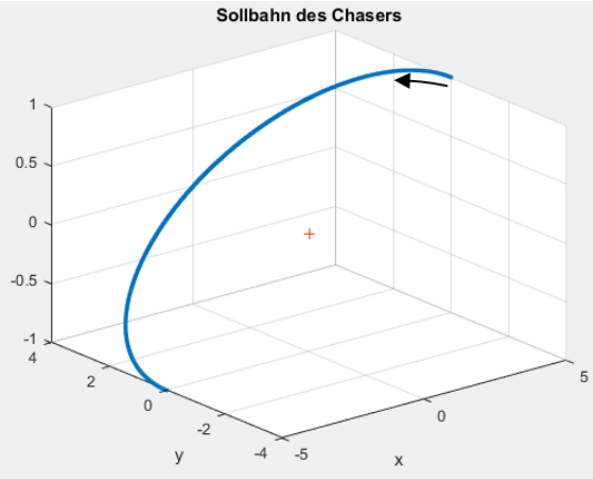



Abb. 7–15: Sollbahn des Chasers Scenario 2

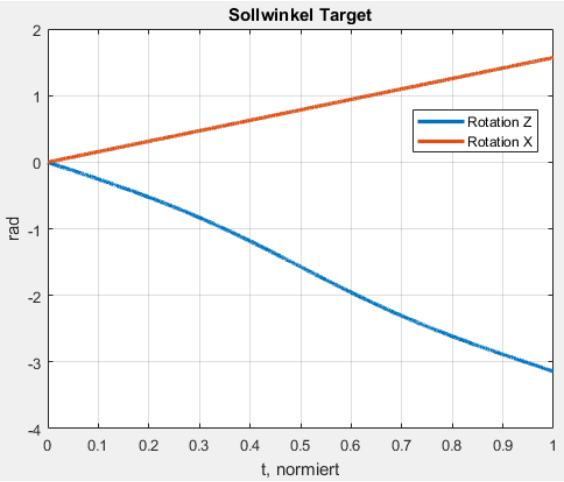



Abb. 7–16: Sollwinkel des Targets Scenario 2

Aus Abb. 7–17 zeigt die oszillierende Bewegung des Chasers auf der x-Achse, die sich wiederum aus der elliptischen Bahn ergibt. Damit sich Chaser und Target auf derselben Ebene befinden, hat die z-Koordinate des Chasers denselben Wert wie der Abstand des Targets senkrecht zur xy-Ebene. Dieser wurde im Vorhinein mit 2m festgelegt und muss bei Benutzung der Hardware zuerst vermessen werden. In Abb. 7–18 werden die dazugehörigen Arbeitsraumkoordinaten dargestellt. Aus Abb. 7–19 lassen sich die Gelenkkoordinaten für das Target entnehmen, aus der zum einen das Erreichen der Grenze für die Target Head Rotation X, zum anderen das Erreichen der Grenze Target C Sled Y ersichtlich wird. Ab diesem Zeitpunkt müssen die anderen Achsen das Wegfallen dieser Freiheitsgrade kompensieren, was sich in einer stärkeren Änderung der Gelenkwinkel äußert. Die schwingende Rotation um die z-Achse in Abb. 7–20 resultiert wiederum aus der vereinfachten Annahme einer konstanten Winkelgeschwindigkeit bei einem elliptischen Umflug. Ebenfalls rotiert das Target wie gewünscht von 0° auf 90° und lässt dabei keine Rotation um die y-Achse zu. Daraus lässt sich folgern, wenn die Freiheitsgrade derart durch die physischen Grenzen eingeschränkt werden, wird es irgendwann den verbleibenden Achsen nicht mehr möglich sein, diese zu kompensieren, da sie selbst an ihre Grenzen stoßen werden. Die durchschnittlich benötigte Rechenzeit beträgt 16 ms. Die Auswertung des Fehlers ist aufgrund der nichtvorhandenen Lösung gegen Ende der Simulation nicht möglich.

Die wichtigere Information ist die, dass die Anlage an ihre Grenzen kommt und es bei dieser Konfiguration keine Lösung mehr gibt. Gekoppelte kontinuierliche Rotationen um mehrere Achsen stellen offensichtlich zum einen für die Hardware mit dessen physischen Grenzen, zum anderen für den Algorithmus Probleme dar. Für weiterführende Arbeiten kann man versuchen die Gewichtungsfunktion anzupassen, indem man die Gewichtung schneller ansteigen lässt und diese sich mehr an einer Parabel orientiert als an einer „Badewanne“.

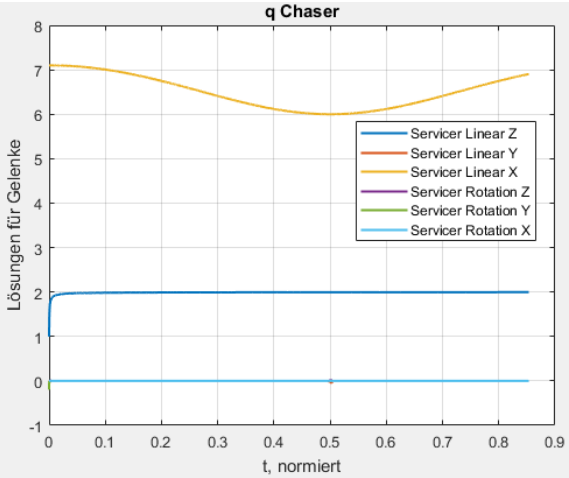



Abb. 7–17: Gelenkkoordinaten q für den Chaser Scenario 2

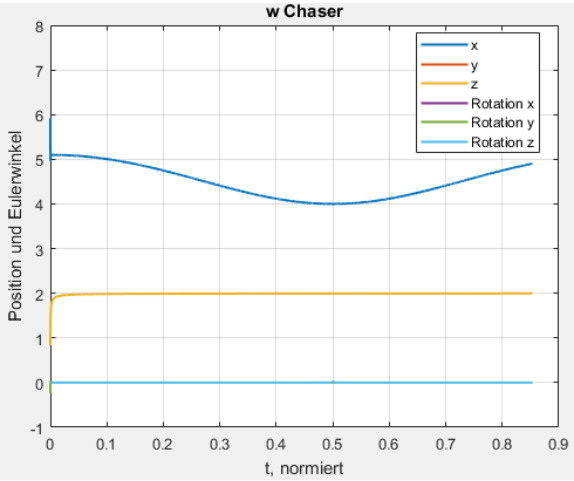



Abb. 7–18: Arbeitsraumkoordinaten w für den Chaser Scenario 2

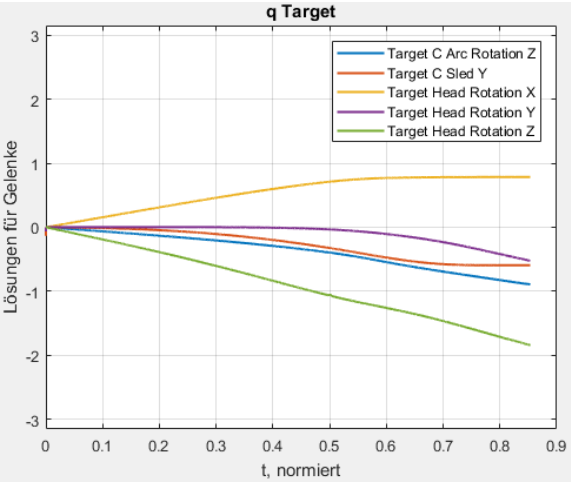



Abb. 7–19: Gelenkkoordinaten q für das Target Scenario 2

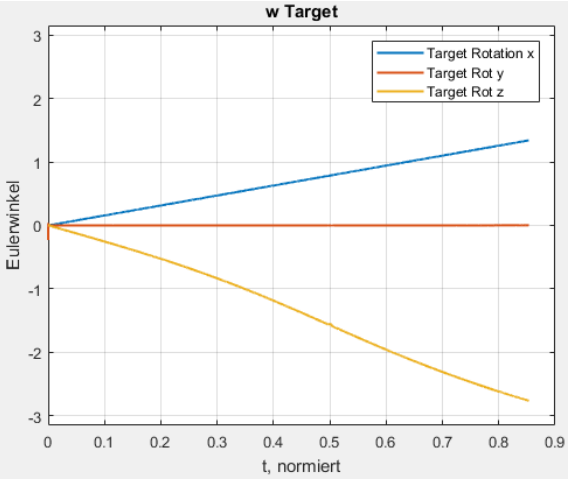



Abb. 7–20: Arbeitsraumkoordinaten w für das Target Scenario 2

### 7.1.3 Scenario 3

Nach dem extremen Beispiel, bei dem die Anlage und der Algorithmus an seine Grenzen stößt, behandelt das dritte Scenario eine weniger komplexe Situation. Der Chaser fliegt wieder um das Target und das Target besitzt feste Ausgangslagen für die Rotation um die x- und um die y-Achse. Zusätzlich soll sich das Target entgegen der Bewegungsrichtung des Chasers um insgesamt -90° um die z-Achse drehen. Damit ergibt sich für das Target eine Endausrichtung von −90°, 360° bzw. 0° aus dem kompletten Umflug des Chasers, sowie weiteren -90° aus der Eigendrehung. Allgemeine Daten können aus Tab. 7–4 entnommen und die Sollbahn des Chasers bzw. die Sollwinkel für das Target in Abb. 7–21 und Abb. 7–22 betrachtet werden.

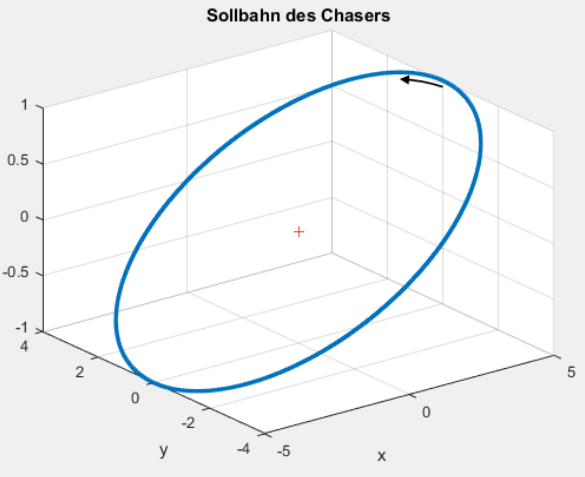



Abb. 7–21: Sollbahn des Chasers Scenario 3

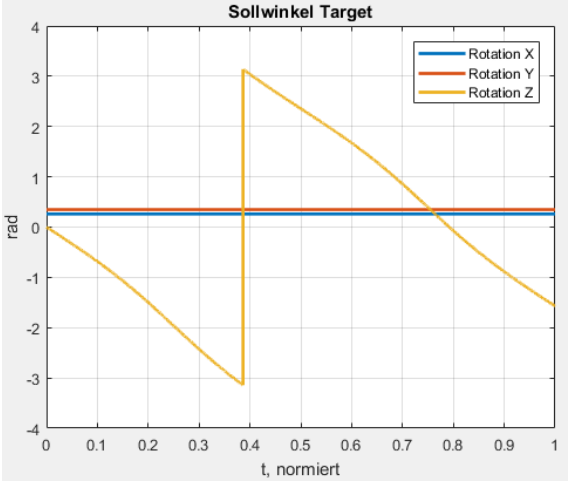



Abb. 7–22: Sollwinkel des Targets Scenario 3

Tab. 7–4: Allgemeine Daten zu Scenario 3

| Parameter | Wert |
|---|---|
| **T** | 10ms |
| **dt** | 2.5ms |
| **ω** | 0.03°/10ms |
| **Target C-Sled Gewichtung** | 5 |
| **Prozessor** | Intel® Xeon® CPU E3-1270 v3 @ 3.50 GHz |
| **Punkte gesamt** | 12001 |

Die Ausgangslage für das Target ist eine Rotation von 20° um die y-Achse und eine Rotation um 15° um die x-Achse, die über die gesamte Zeit konstant bleiben sollen. Die Auswertung der Berechnung kann aus Tab. 7–5 entnommen werden. Es zeigt sich wieder eine Unstetigkeit, wenn der Chaser um 180° um das Target geflogen ist (vgl. Abb. 7–23 und Abb. 7–24 für t=0.5). Für die Hardware würde das nur ein Ruckeln bedeuten, da die nächste Position wieder dem Sollverlauf entspricht. Für das Target ergeben sich kontinuierliche Verläufe der Gelenkkoordinaten, die die gewünschten Drehungen im Arbeitsraum erzeugen (vgl. Abb. 7–25 und Abb. 7–26). Das stetige Verfahren aller Achsen resultiert daraus, dass die Gelenke global auf alle Eulerwinkel der Anlage Einfluss haben. In anderen Worten, die Lösungen für die Gelenke lassen sich nicht superponieren und als endgültigen Winkel im Arbeitsraum festlegen. Die „Target C Arc Rotation Z“, die um die globale z-Achse erfolgt, ändert beispielsweise bei einem gekippten Satelliten nicht nur den Winkel γ, sondern auch die Winkel β und α. Das lässt sich nachvollziehen, wenn man sich die kinematische Kette des Systems vor Augen führt.

Die vorgegebene Berechnungszeit von 10 ms wurde im Schnitt um 1.6 ms überzogen und der durchschnittliche Fehler lag bei 0.0066. Der maximale Fehler von 1.752 stammt aus der fehlerhaften Berechnung bei t=0.5 (vgl. Tab. 7–5). Die Einheit der Fehler ist eine Mischung aus [rad] und [m], da dieser aus der Norm der Differenz

zwischen Soll- und Ist-Position berechnet wird und die Positionen sowohl Ortskoordinaten als auch Winkel beinhalten.

Tab. 7–5: Ergebnisse und Startwerte für Scenario 3

| | |
|---|---|
| Rotation um x | 15° |
| Rotation um y | 20° |
| Max. Fehler | 1.7152 |
| Ø Fehler | 0.0066 |
| Ø Berechnungsdauer | 0.0116 s |

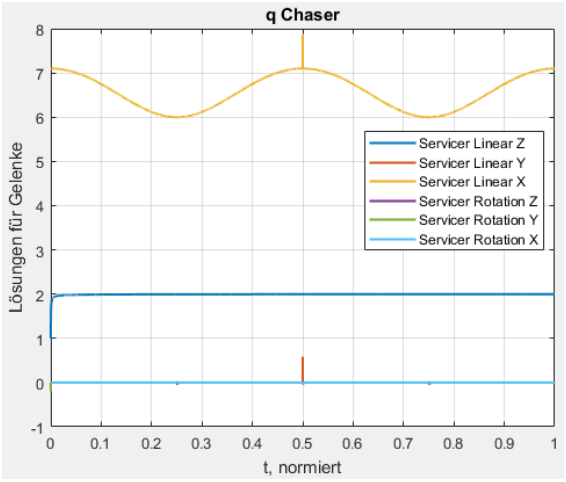



Abb. 7–23: Gelenkkoordinaten q für den Chaser Scenario 3

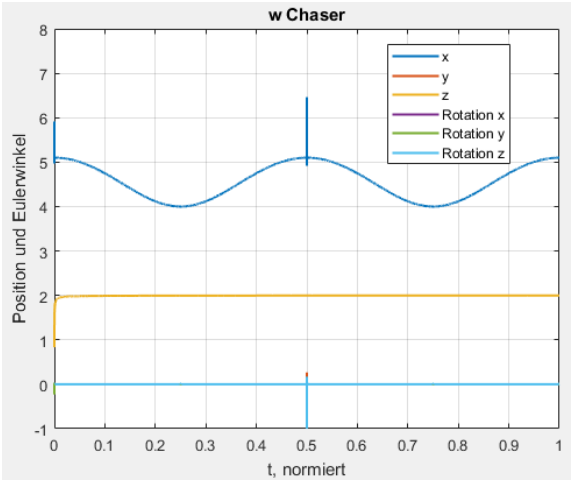



Abb. 7–24: Arbeitsraumkoordinaten w für den Chaser Scenario 3

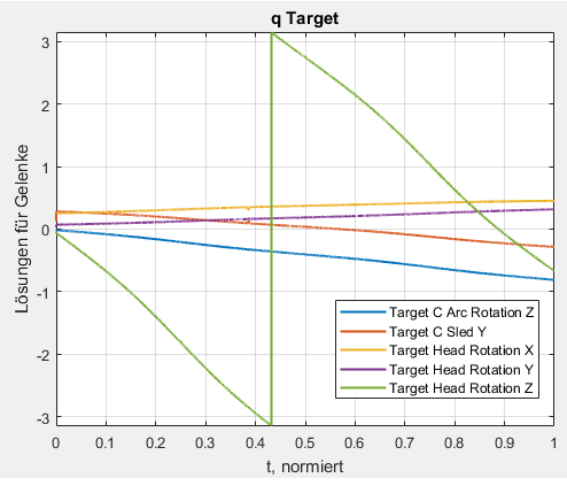



Abb. 7–25: Gelenkkoordinaten q für das Target Scenario 3

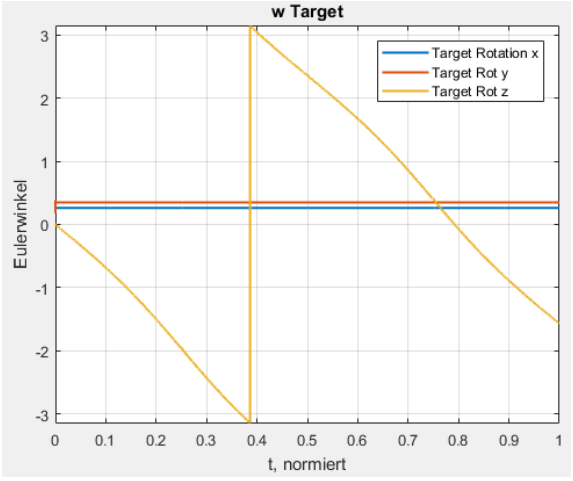



Abb. 7–26: Arbeitsraumkoordinaten w für das Target Scenario 3

Zusammenfassend lässt sich sagen, dass der Algorithmus verschiedene Problemstellungen mit wenig Rechenzeit und geringen Fehler lösen kann. Es gibt singuläre Stellen, die Schwierigkeiten bereiten, jedoch mit einer Verkleinerung des Berechnungsintervalls ebenfalls ohne größere Fehler gelöst werden können. Für den Fall, dass der Roboter sich aus einer Position nicht mehr lösen kann, ohne Gelenke instantan um 180° zu drehen, findet der Algorithmus dann keine Lösung mehr. Dieses Verhalten kann möglicherweise mit einer Veränderung der Gewichtungsfunktionen verbessert werden. Eine wichtige Erkenntnis ist der Trade-Off zwischen Rechenzeit und Genauigkeit, wobei ein Minimum an Rechenzeit, auch wenn diese >10 ms ist, nötig ist, um akzeptable Ergebnisse zu erzielen.

## 7.2 Verifikation im RACOON-Lab

Die Verifikation in der Simulationsumgebung des RACOON-Lab gestaltet sich schwierig, da man die Bewegung der Satelliten nicht mit Plots oder ähnlichem analysieren kann. Dafür muss man den Code kompilieren, die dll-Datei in den Axismapper integrieren und diese dort eigens testen. Um dennoch nachweisen zu können, dass der MATLAB-Code integriert ist, wird eine Ausgangslage beider Satelliten eingegeben und die Ausgabedaten für die einzelnen Gelenke überprüft.

Die Sollwerte für den Algorithmus und die Eingabe in das Terminal Control sind in Tab. 7–6 aufgelistet:

Tab. 7–6: Eingangswerte für die Anlage

| Parameter | Wert | Control Terminal Eingabe |
|---|---|---|
| **X** | -5.574 m | 5 m |
| **Y** | -0.259 m | 0 m |
| **Z** | -0.221 m | 0 m |
| **Rot x Chaser** | 0.1745 = 10° | 10° |
| **Rot y Chaser** | 0.3490= 20° | 20° |
| **Rot z Chaser** | 0.5235 = 30° | 30° |
| **Rot x Target** | 0.1745= 10° | 10° |
| **Rot y Target** | 0.3490 = 20° | 20° |
| **Rot z Target** | 0.5235= 30° | 30° |

Die Eingabedaten sind alle in Base bzw. ECI-Koordinaten zu tätigen. Das Berechnungsintervall für den Algorithmus beträgt dt=0.0025 s. Im Vorhinein lässt sich sagen, dass die Gelenkwinkel nicht identisch mit der Eingabe sein werden, da der Ortsvektor eine y- und z-Komponente und somit alle Vektoren gedreht werden müssen. In den Versuchen wurde festgestellt, dass die Eingabe der Winkelgeschwindigkeit um die z- und x-Achse vertauscht sind. Bei einer Eingabe für eine Drehung um die z-Achse rotiert der Satellit anstatt dessen um die x-Achse. Der Eingabevektor für den Algorithmus ist bei einer positiven Eingabe in das Control

Terminal negativ. Damit die Daten konsistent für die Anlage sind, werden diese im Algorithmus mit −1 multipliziert.

Die Ausgabe des Algorithmus ist in Tab. 7–7 zu sehen:

Tab. 7–7: Position des Roboters aus der Berechnung des Algorithmus'

| Parameter | Wert | Control Terminal Eingabe |
|---|---|---|
| **X** | 5.595 m | 5 m |
| **Y** | -0.009 m | 0 m |
| **Z** | 0.999 m | 0 m |
| **Rot x Chaser** | 0.2369 = 13.57° | 10° |
| **Rot y Chaser** | 0.3759 = 21.54° | 20° |
| **Rot z Chaser** | 0.6820 = 39.07 ° | 30° |
| **Rot x Target** | 0.2645 = 15.15° | 10° |
| **Rot y Target** | 0.3913 = 22.42 ° | 20° |
| **Rot z Target** | 0.6457 = 36.99 ° | 30° |

Die Berechnung der Gelenkkoordinaten ergab folgende Werte:

Tab. 7–8: Lösung für die Gelenkkoordinaten

| Freiheitsgrad | Wert |
|---|---|
| **Servicer Linear Z** | 1.0 m |
| **Servicer Linear Y** | -0.442 m |
| **Servicer Linear X** | 6.504 m |
| **Servicer Rotation Z** | 0.4460 = 25.55° |
| **Servicer Rotation Y** | 0.3481 = 19.94° |
| **Servicer Rotation X** | 0.0578 = 3.31° |
| **Target C Arc Rotation Z** | 0.0303 = 1.73 ° |
| **Target C Sled Y** | 0.2239 = 12.83° |
| **Target Head Rotation X** | 0.0869 = 4.98° |
| **Target Head Rotation Y** | 0.1271 = 7.28° |
| **Target Head Rotation Z** | 0.4010 = 22.97° |

Die Berechnungsdauer in Visual Studio betrug 18.89 ms und man erkennt noch kleinere Abweichungen zu der Sollvorgabe. Für die Übergabe der Parameter an die Inverse Kinematik Berechnung gibt es Verbesserungspotenzial, da die Werte einzeln als double übergeben werden. Aus unbekannten Gründen wird die alte Sollposition nicht abgespeichert, was zu dem relativ großen Fehler in der Berechnung geführt

haben könnte. Wenn man die Ausrichtung in z von Chaser und Target vergleicht, gibt es einen Unterschied von ca. 2 ° bzw. von 5.13 %.

Wenn die Übergabe der Daten besser erfolgt und der Matlab Code stabiler integriert ist, z.B. durch Anlegen einer lokalen Variable, die durch den Algorithmus nur noch über die Iteration die Gelenkkoordinaten, Sollpositionen und die Position des Roboters aktualisiert.

# 8 Zusammenfassung

Das RACOON-Lab setzt sich aus einem Chaser mit drei translatorischen und 3 rotatorischen Freiheitsgraden sowie einem Target mit 5 rotatorischen Freiheitsgraden zusammen. Die Simulationsumgebung soll es ermöglichen in Echtzeit relative Satellitenbewegungen darzustellen. Deshalb wurde in dieser Arbeit die inverse Kinematik für dieses System ausgelegt. Inverse Kinematik bedeutet, über eine geeignete Beschreibung der Anlage und der Sollposition, die Gelenkwinkel bzw. –koordinaten zu berechnen.

Der Arbeitsraum, der die absolute Lage bestimmt, wird gebildet durch die Eulerwinkel beider Satelliten und deren örtlichen relativen Lage zueinander. Da das System mehr Freiheitsgrade (11) als Arbeitsraumkoordinaten (9) besitzt, muss ein geeigneter Zusammenhang zwischen diesen erstellt werden. Vorher musste jedoch eine geeignete Beschreibung der Koordinatensysteme stattfinden, die an die DH-Parametrisierung angelehnt ist und sich nur geringfügig unterscheidet. Dabei wird das inertiale Koordinatensystem durch Verdrehungen um die x- bzw. z-Achse sowie Verschiebungen in diese Richtungen so lange gedreht bis die z-Achse mit dem nächsten Freiheitsgrad der kinematischen Kette übereinstimmt. Am Ende dreht man das Koordinatensystem wieder zum inertialen Koordinatensystem. Mit Hilfe der resultierenden DH-Parameter ergeben sich absolute und relative Koordinatentransformationen, die die Beziehung zwischen Gelenk und Basis bzw. zwischen zwei Gelenken widerspiegelt. Über eine Gewichtungsmatrix und der Jakobimatrix, die aus der Beschreibung des Systems resultiert und widerspiegelt, welcher Freiheitsgrad welchen Einfluss auf die Arbeitsraumkoordinaten hat, kann man die gesuchten Gelenkwinkel bzw. –koordinaten berechnen.

Nach der Abstrahierung und der mathematischen Formulierung des Problems auf Geschwindigkeitsebene werden über die Gewichtungsmatrix die physischen Systemgrenzen eingebunden. Der Algorithmus wird die Gewichtung eines Freiheitsgrades bis ins Unendliche erhöhen, wenn sich der Roboter einer Grenze jenes Freiheitsgrades nähert. Das hat zur Folge, dass dem Roboter dieses Gelenk nicht mehr zur Verfügung steht und die Bewegung mit den anderen Freiheitsgraden ausgleichen muss. Die Beschreibung der Winkel erfolgt mittels Euler, die man mit der Atan2-Funktion aus den Rotationsmatrizen berechnen kann. Diese hat die negative Eigenschaft einer Singularität bei ±90°, da man zur Berechnung zweier Winkel durch den Kosinus des anderen Winkels und somit durch null teilt. Um das Problem zu umgehen, muss anfangs das inertiale Koordinatensystem in der Nähe dieser Singularität um 90° gedreht und anschließend nach der Berechnung die Gelenkwinkel und Koordinaten wieder auf die richtige Basis übertragen werden. Eine zusätzliche Eigenschaft des Roboters soll es sein, seine Freiheitsgrade über 360° hinaus drehen zu können. Dafür werden die Eingangsdaten, die Position des Roboters und die alte Sollposition miteinander verglichen und dann so angepasst, dass dieser sich über den Randbereich von ±180° bewegen kann. Diese Anpassung erfolgt intern im Algorithmus, wobei die Ausgabe wieder in der Eulerwinkeldarstellung gemacht wird.

Neben den systematischen Problemen, stellten sich noch Schwierigkeiten bei der Kompilierung des Programms heraus. Der anfangs verwendete MATLAB Coder brachte nur mangelhafte Ergebnisse hervor, die mit einem hohen Aufwand an Umprogrammierung verbunden waren. Deshalb wurde das Programm mit dem Library Compiler kompiliert, der bessere Ergebnisse erzielt hat und deutlich schneller war.

Bevor es in Visual Studio integriert wird, musste die inverse Kinematik noch in MATLAB getestet werden, das zur Programmierung deshalb ausgewählt wurde, weil die Verifikation sich dort als einfach gestaltet. Die Ergebnisse zeigen eine breite Anwendbarkeit des Algorithmus, der zuverlässig und schnell die beste Konfiguration an Gelenkwinkel bzw. -koordinaten berechnet. An den kritischen Stellen gibt es zwar leichte Abweichungen, die aber mit einem geringeren Berechnungsintervall weiter minimiert werden können. Der Algorithmus hat Schwierigkeiten eine geeignete Lösung zu finden, wenn der Roboter seine verfügbaren Freiheitsgrade voll ausgenutzt hat und die neue Position nur erreicht werden kann, wenn ein Freiheitsgrad um 180° gedreht wird. In anderen Worten, wenn der Roboter seine physischen Grenzen erreicht hat, wird es schwer eine akzeptable Lösung zu finden. Das kann aber möglicherweise mit einer Veränderung der Gewichtungsmatrix optimiert werden, sodass der Roboter nie in so eine Position fährt, aus der kein kontinuierliches Weiterfahren mehr möglich ist.

Der letzte Schritt war die Einbindung des MATLAB-Skripts in Visual Studio und der Test in der Simulationsumgebung. Nach den erfolgreichen Tests in MATLAB, stellten sich die Versuche dort als nicht optimal heraus. Zusätzlich ist zu bemerken, dass der Code für Visual Studio umgeändert werden musste, damit dieser im Axismapper ausgeführt werden konnte. Dabei gehen bisher wichtige Informationen verloren, da die Kommunikation zwischen dem MATLAB-Skript und dem Axismapper nicht reibungsfrei stattfindet. Der Roboter wird bei jeder Iteration neu initialisiert, was zur Folge hat, dass man Daten lokal zwischenspeichern muss, da man diese nicht in der Klasse Roboter überschreiben kann. Für einfachere Szenarien hat die Simulation in Visual Studio funktioniert, jedoch bringt der Axismapper irgendwann schlechte Ergebnisse hervor. Für die Hardware würden kleine Fehler nur ein Ruckeln bedeuten, in der Softwaresimulation werden diese aber genau abgebildet, ohne die maximalen Verfahrgeschwindigkeiten des Roboters zu berücksichtigen.

Das Ergebnis dieser Arbeit ist eine eindeutige Beschreibung der inversen Kinematik des RACOON-Lab, die robust und mit Möglichkeiten zur Erweiterung bzw. Verbesserung in MATLAB implementiert wurde. Die optimale Einbindung in Visual Studio stellt jetzt noch den letzten wichtigen Schritt dar, um umfangreiche Simulationen mit dem RACOON-LAB zu ermöglichen.

# 9 Literatur

A. Fleischner, M. Wilde and U. Walter (2012), 'RACOON - A Hardware In-The-Loop Simulation Environment For Teleoperated Proximity Operations', *I-SAIRAS, Turin*, 2012.

Craig, J. J. (2014), *Introduction to robotics*: *Mechanics and control* (3. ed., new internat. ed., Harlow: Pearson; Pearson Education).

Lehrstuhl für Raumfahrttechnik: RacoonLab, 'RACOON Lab' <http://www.lrt.mw.tum.de/index.php?id=54>, accessed 5 Jul 2017.

Pfeiffer, F., and Reithmeier, E. (1987), *Roboterdynamik*: *Eine Einführung in die Grundlagen und technischen Anwendungen* (Teubner-Studienbücher, Stuttgart: Teubner).

Pfeiffer, F., and Schindler, T. (2014), *Einführung in die Dynamik* (3. Aufl., Berlin: Springer Vieweg).

Shigley, J. E., and Uicker, J. J. (1995), *Theory of machines and mechanisms* (McGraw Hill series in mechanical engineering; 2. ed., New York: McGraw Hill).

Siciliano, B., Oriolo, G., Sciavicco, L. et al. (2009), *Robotics*: *Modelling, Planning and Control* (Advanced Textbooks in Control and Signal Processing, London: Springer London).

Singla P., Mortari D. and Junkins J. L. (2005), 'How to Avoid Singularity When Using Euler Angles?', *AIAA/AAS Space Flight Mechanics Meeting*, 2005.

Whitney, D. (1969), 'Resolved Motion Rate Control of Manipulators and Human Prostheses', *IEEE Trans. Man Mach. Syst.*, 10/2: 47–53.