

Haptische Exploration und Modelbildung unbekannter Mechanismen

vorgelegt von

Yu Jen Chen

EDV.Nr.:902273

dem Fachbereich VII – Elektrotechnik – Mechatronik – Optometrie
der Berliner Hochschule für Technik Berlin

vorgelegte Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Engineering (B.Eng.)

im Studiengang

Humanoide Robotik

Tag der Abgabe 22. März 2023



Studiere Zukunft

Betreuer

Prof. Dr. Manfred Hild Berliner Hochschule für Technik

Gutachter

Dr. Simon Untergasser Berliner Hochschule für Technik

Kurzfassung

Ziel dieser Arbeit ist die Entwicklung eines Roboters auf der Grundlage eines kinematisch freien Modells, d. h. eines Roboters ohne Kenntnis seiner eigenen Mechanismen, der die Umgebungen mithilfe von sensorlosem haptischen Feedback explores, eine Karte erstellt und die explored Umgebungen identifiziert. Ein fünfgliedriger-Koppelgetriebe-Robotermanipulator mit zwei Freiheitsgraden wurde entwickelt, um die Labyrinthe als unbekannte Umgebungen zu erkunden. Die strombasierte Kollisionserkennung wurde als haptisches Feedback verwendet. Zur Steuerung des Roboters wird ein virtueller mobiler Roboter in einem Configuration Space gebildet und durch den Minimal Recurrent Controller, ein rückgekoppeltes neuronales Netzwerk aus zwei Neuronen, gesteuert. In Ermangelung eines kinematischen Modells führt das Modell der vom Roboter erstellten Karte zu dem Ergebnis, dass der Roboter die unbekannte Umgebung vollständig erkundet hat und in der Lage ist, verschiedene Umgebungen bis zu einem gewissen Grad zu erkennen. Dies zeigt, dass es möglich ist, das Verhalten des Roboters zu steuern, ohne ihm ein kinematisches Modell zu geben. Dies kann ohne maschinelle Lernmethoden oder Optimierungssteuerung erreicht werden.

Abstract

The aim of this work is to develop a robot based on a kinematic-free model, i.e., a robot without knowledge of its mechanisms, which explores the environments using sensor less haptic feedback, creates a map and identifies the explored environments. A five-bar-linkage-robot-manipulator with two degrees of freedom was developed to explore mazes as unknown environments. Current-based collision detection was used as haptic feedback. To control the robot, a virtual mobile robot is formed in Configuration Space and controlled by Minimal Recurrent Controller, a feedback neural network of two neurons. In the absence of a kinematic model, the model of the map created by the robot leads to the result that the robot has fully explored the unknown environment and is able to recognize different environments to some extent. This proves that it is possible to control the robot's behavior without giving it a kinematic model. This can be achieved without machine learning methods or optimization control.

Erklärung

Ich versichere, dass ich diese Abschlussarbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Datum

Unterschrift

Inhaltsverzeichnis

1 Einleitung	3
1.1 Problemstellung	3
1.2 Stand der Forschung	4
1.3 Ziel der Arbeit	4
1.4 Aufbau der Arbeit	5
2 Beschreibung der verwendeten Roboter	7
2.1 Überblick über den Roboter	7
2.2 Komponenten des Roboters	8
2.2.1 Motoren	8
2.2.2 Monitor	8
2.2.3 Mikrocontroller	9
2.2.4 Mechanische Konfiguration für Modus 1 und Modus 2	9
3 Thematische Grundlagen und praktische Umsetzungen	11
3.1 Die theoretischen Grundlagen von Modus 1 und Modus 2	11
3.1.1 Freiheitsgrad	11
3.1.2 Configuration	12
3.1.3 Configuration Space	12
3.2 Die theoretischen Grundlagen von Modus 1	13
3.2.1 PID-Regler	13
3.3 Die theoretischen Grundlagen von Modus 2	14
3.3.1 Work Space	14
3.3.2 Singularität	16
3.3.3 Minimal Recurrent Controller	17
3.3.4 Moment in Bildverarbeitung	17
4 Modus 1	19
4.1 Verwendungszwecke des Modus 1	19
4.2 Überblick über den Algorithmus	20
4.3 Detaillierte Beschreibung des Algorithmus	20
4.4 Ergebnisse	21
4.5 Zusammenfassung und Ausblick	24
5 Modus 2	25
5.1 Verwendungszwecke des Modus 2	25
5.2 Überblick über den Algorithmus	26
5.3 Detaillierte Beschreibung des Algorithmus	28
5.3.1 Exploration	28
5.3.2 Kartenvergleich	30
5.4 Ergebnisse	32

5.4.1	Exploration	32
5.4.2	Kartenvergleich	33
5.5	Weitere Variante	40
5.6	Zusammenfassung und Ausblick	40
Literatur- und Quellenverzeichnis		42
A Anhang		45
A.1	Zusätzliche Ergebnisse von Modus2	45
A.1.1	Beispiele für andere Kartenvergleich	45
A.1.2	Alle in 45-Grad-schrittweise erstellte Karten	48
B.2	Quellcode für Modus 1	49
C.3	Quellcode für Modus 2	52
C.3.1	Exploration	53
C.3.2	Kartenvergleich	58

Abbildungsverzeichnis

2.1	Schaltplan der Hauptplatine des Roboters	7
2.2	Servomotor XL330-M077-T von <i>ROBOTIS Co., Ltd.</i>	8
2.3	2.4-Inch-LCD-Modus 1 <i>Waveshare Electronics</i>	8
2.4	OpenCM9.04 typeC Mikrocontroller	9
2.5	3D-Modell und Technische Zeichnung bei Modus 1	10
2.6	3D-Modell und Technische Zeichnung bei Modus 2	10
2.7	Aufbau des Roboters bei Modus 1 (links) und Modus 2 (rechts)	10
3.1	Schematische Darstellung der Mechanismen bei Modus 1 (links) und Modus 2 (rechts)	11
3.2	<i>Work Space</i> der zwei-Drehgelenke-in-Reihe	14
3.3	<i>Work Space</i> des auf fünfgliedrigem Koppelgetriebe basierenden Roboters	15
3.4	5 Typen der Konfigurationen eines Roboters mit fünfgliedrigem Koppelgetriebe	15
3.5	Singularität des fünfgliedrigen Koppelgetriebes	16
3.6	MRC Neuronales Netzwerk	17
4.1	Schematische Darstellung des Roboters bei Modus 1	19
4.2	Der Monitor am Roboter bei Modus 1	19
4.3	Schematische Darstellung des <i>C Space</i> bei Modus 1	20
4.4	Schematische Darstellung zum Modus 1 Algorithmus	21
4.5	Visuelle Darstellung des <i>C Space</i> bei Modus 1	21
4.6	Visuelle Darstellung des <i>C Space</i> bei Modus 1	22
4.7	Visuelle Darstellung des <i>C Space</i> bei Modus 1	22
4.8	Strom, Kollisionserkennung und <i>Configuration</i> über Zeit-Diagramm	23
4.9	Visuelle Darstellung des <i>C Space</i> bei Modus 1	23
5.1	Schematische Darstellung des Roboters bei Modus 2	25
5.2	Prozess bei Modus 2 als Flussdiagramm	25
5.3	Virtueller mobiler Roboter <i>R</i> im eingeschränkten <i>C Space</i>	27
5.4	Der Fall von Mehrfachlösungen eines fünfgliedrigen Koppelgetriebes	27
5.5	Übersicht über den MRC-basierten haptischen Explorationsalgorithmus	28
5.6	Ergebnisse der Exploration	33
5.7	Ergebnisse der Exploration	33
5.8	Schematische Darstellung der Beziehung zwischen <i>Work Space</i> und <i>C Space</i> des Roboters	34
5.9	Die Labyrinth in der echten Welt in Gruppe 1	34
5.10	Die heruntergetasteten und markierten Karten in Gruppe 1(links Karte 1 <i>K</i> ₁ , rechts Karte 2 <i>K</i> ₂)	35
5.11	Die normalisierte und nicht-normalisierte Summe des minimalen euklidischen Abstands über Rotation von 0 [°] bis 360 [°]	35
5.12	Die Labyrinth in der echten Welt in Gruppe 2	36

5.13 Die heruntergetasteten und markierten Karten in Gruppe 2(links Karte 1 K_1 , rechts Karte 2 K_2)	36
5.14 Die normalisierte und nicht-normalisierte Summe des minimalen euklidischen Abstands über Rotation von 0° bis 360°	36
5.15 Die Labyrinththe in echter Welt in Gruppe 3	38
5.16 Die heruntergetasteten und markierten Karten in Gruppe 3(links Karte 1 K_1 , rechts Karte 2 K_2)	38
5.17 Die normalisierte und nicht-normalisierte Summe des minimalen euklidischen Abstands über Rotation von 0° bis 360°	38
5.18 Eine der möglichen Varianten des Roboters bei Modus 2 und der entsprechende $C Space$	40

Tabellenverzeichnis

5.1	Feature Set von Karte 1 in Gruppe 1	35
5.2	Feature Set von Karte 2 in Gruppe 1	35
5.3	Feature Set von Karte 1 in Gruppe 2	37
5.4	Feature Set von Karte 2 in Gruppe 2	37
5.5	Feature Set von Karte 1 in Gruppe 3	39
5.6	Feature Set von Karte 2 in Gruppe 3	39
5.7	Resultierender MAPE und Standardabweichung	39

Kapitel 1

Einleitung

In einem dunklen Raum verlassen sich Menschen in der Regel auf haptische Wahrnehmung, um die Umgebung zu erkennen und zu erkunden, wie zum Beispiel, indem sie Wände berühren, die Position von Hindernissen spüren oder nach dem Lichtschalter suchen. In der Robotik ist die Umgebungsexploration ebenfalls ein wichtiger Forschungsbereich, da es notwendig ist, vor oder während der Aufgabenerfüllung ein Verständnis der Umgebung aufzubauen. Dies hilft, Hindernisse zu vermeiden, Schäden zu reduzieren und die Effizienz der Aufgabenerfüllung zu verbessern.

Um intelligenter Roboter zu entwickeln, die wie Menschen ihre Umgebung erkunden können, ist es lohnenswert zu erforschen, wie man ohne visuelle Unterstützung mit der haptischen Wahrnehmung eine Umgebung erkunden kann. Diese Fähigkeit kann dazu beitragen, dass Roboter flexibler sind und in schlecht beleuchteten Umgebungen arbeiten können, während sie auch helfen können, dass Roboter sich besser an unterschiedliche Umgebungen anpassen können.

Daher kann diese Fähigkeit des Menschen als Referenz und Inspiration für die Robotik dienen. Durch die Nachahmung menschlichen Verhaltens und das Lernen aus den Erfahrungen von Robotern können intelligente und effizientere Roboter entwickelt werden, um eine breitere Anwendung zu erreichen.

1.1 Problemstellung

Im Bereich der Robotik liegt es viele Wahrnehmungsmethoden vor, die ähnlich wie menschliche Wahrnehmung funktionieren. Es ist üblich, Sensoren wie LiDAR oder optische Kameras zur Wahrnehmung der Umgebung von Robotern einzusetzen. Im Vergleich dazu gibt es jedoch einige unersetzbare Merkmale der auf haptisches Feedbacks basierenden Wahrnehmungsmethoden: Zum Beispiel ist haptisches Feedback im Gegensatz zu LiDAR in der Regel nicht durch das Material des zu erkennenden Objekts (wie z. B. Glas) eingeschränkt oder kann eine Kamera durch Umgebungslicht beeinträchtigt werden und ihre Funktion verlieren. Die Methoden zur haptischen Wahrnehmung kann in sensorische oder sensorlose unterteilt werden. Im Vergleich zu sensorlosem Ansatz kann sensorischer Ansatz präzisere und reichhaltigere Informationen liefern. Sensorloses haptisches Feedback ist jedoch wirtschaftlicher und wird daher auch untersucht.

Wie bereits erwähnt, erkunden Menschen die Umgebung, indem sie ihre Gliedmaßen ausstrecken. Bei der Anwendung von Robotern kann berücksichtigt werden, wie humanoide Roboter bzw. Manipulatoren sich bewegen und wohin sie sich bewegen sollten, um die Umgebung zu erkunden. Um die Bewegung des Roboters zu steuern, ist es oft notwendig, das kinematische Modell des Roboters zu haben, was bedeutet, dass komplexe mathematische Modelle für Vorwärts- und

Rückwärtsskinematik erstellt werden müssen. Dies erhöht nicht nur die Komplexität des Algorithmus, sondern erfordert auch einen höheren Aufwand des Controllers.

Nachdem die Menschen die Umgebung erkundet haben, erstellen sie oft ein Modell der Umgebung und wissen beispielsweise, wo sich Lichtschalter oder Hindernisse befinden. In ähnlicher Weise müssen Roboter bei der Erkundung einer unbekannten Umgebung ihre Umgebung verstehen, einschließlich der räumlichen Struktur der Umgebung, der Lage und Form von Hindernissen usw.

Unter den oben genannten sensorischen und steuerungsbezogenen Aspekten wäre es möglich, die Notwendigkeit einer haptischen Exploration der Umgebung ohne den Einsatz von Sensoren und kinematischen Modellen zu überwinden und ein gewisses Maß an Verständnis für die Umgebung zu entwickeln, was zu einem theoretisch und wirtschaftlich schlankeren Ansatz für die Exploration führt.

1.2 Stand der Forschung

Laut [16, 22] können die Strategien zur Exploration der Umgebung generell in zwei Typen unterteilt werden: *model-based* und *reactive*. Erstere werden häufig zur Lösung von Erkundungsproblemen bei SLAM(Simultaneous localization and mapping) und auch auf Robotermanipulatoren basierende Exploration der Umgebung[20, 26] eingesetzt, wobei die Grundidee darin besteht, den am wenigsten erkundeten Bereich mit Methoden *Information Theory* [25, 23] oder *Frontier Based Exploration* [4, 28] zu explorieren. Letztere hingegen koppeln Wahrnehmung und Aktion direkt miteinander, wie z. B. *Braitenberg Vechicle*[3], *Minimal Recurrent Controller*[19, 11] oder der gängige Algorithmus zur Erkundung von Labyrinthen *Wall follower*[15]. Erstere wird zwar seltener bei der Steuerung von Robotermanipulatoren eingesetzt, aber es lohnt sich zu untersuchen, wie es hier angewendet werden kann, da es reaktionsfähiger und einfacher zu implementieren ist.

Für die sensorlose haptische Wahrnehmung bzw. Kollisionserkennung bei Robotermanipulatoren wurden verschiedene Ansätze entwickelt, z. B. [10, 5]. In der Arbeit von Bethge [2] ist ein Ansatz zur Kollisionserkennung durch künstlicher neuronaler Netze vorgestellt. Darüber hinaus wird die Impedanzkontrolle zur Steuerung eines Roboters in [13] verwendet, der ein Labyrinth mithilfe dem sensorlosen haptischen Feedback exploriert.

Ferner wurden modellfreie Steuerungsmethoden untersucht, die jedoch in der Regel den Einsatz von maschinellem Lernen erfordern, was Zeit und eine Trainingsmenge benötigt [29].

1.3 Ziel der Arbeit

Um einen möglich schlanken und auf haptischem Feedback basierenden Explorationsverfahren zu ermöglichen, ist in dieser Arbeit ein Explorationsalgorithmus für den Robotermanipulator auf einem sensorlosen haptischen Feedback und insbesondere auf unbekannten kinematischen Modell zu entwickeln, d. h. „der Roboter kennt seinen eigenen Mechanismus nicht“ Der Algorithmus wird auf einem Robotermanipulator, bestehend aus zwei Servomotoren, implementiert. Der Roboter wird so konzipiert, dass er in zwei Modi arbeiten kann. Modus 1 konzentriert sich auf den Bau eines einfachen Teleroboters mit dem Ziel, die Steuerungsmethode in Modus 2 ohne ein kinematisches Modell und sensorlose Kollisionserkennung zu erweitern. In Modus 2 wird ein fünfgliedriges Koppelgetriebe verwendet, um das Labyrinth zu erkunden, Karten der unbekannten Umgebung zu modellieren und diese zu vergleichen.

1.4 Aufbau der Arbeit

Der Aufbau des gesamten Aufsatzes ist wie folgt: Alle relevanten technischen Details und das Hardware-Design werden in Kapitel 2 vorgestellt. Es folgt Kapitel 3, in dem die Grundlagen der Anwendung der beiden Modi vorgestellt werden. In den Kapiteln 4 und 5 werden ein Überblick und eine detaillierte Beschreibung der Algorithmen für die Modi 1. und 2. gegeben, zusammen mit den Ergebnissen und einer Zusammenfassung bzw. einem Ausblick.

Kapitel 2

Beschreibung der verwendeten Roboter

Im Folgenden wird der in dieser Arbeit verwendeten Roboter vorgestellt. Es werden der Schaltplan, die wichtigsten verwendeten Komponenten und die Hardwarekonfiguration für die beiden Modi beschrieben.

2.1 Überblick über den Roboter

Der Roboter besteht aus zwei Servomotoren, einer Hauptplatine, einem Mikrocontroller, vier Tasten, einem Monitor und zwei verschiedenen Hebeln. Es folgt ein Schaltplan des gesamten Systems. Der Motor ist in dem Plan nicht markiert, da er direkt mit dem Mikrocontroller verbunden ist.

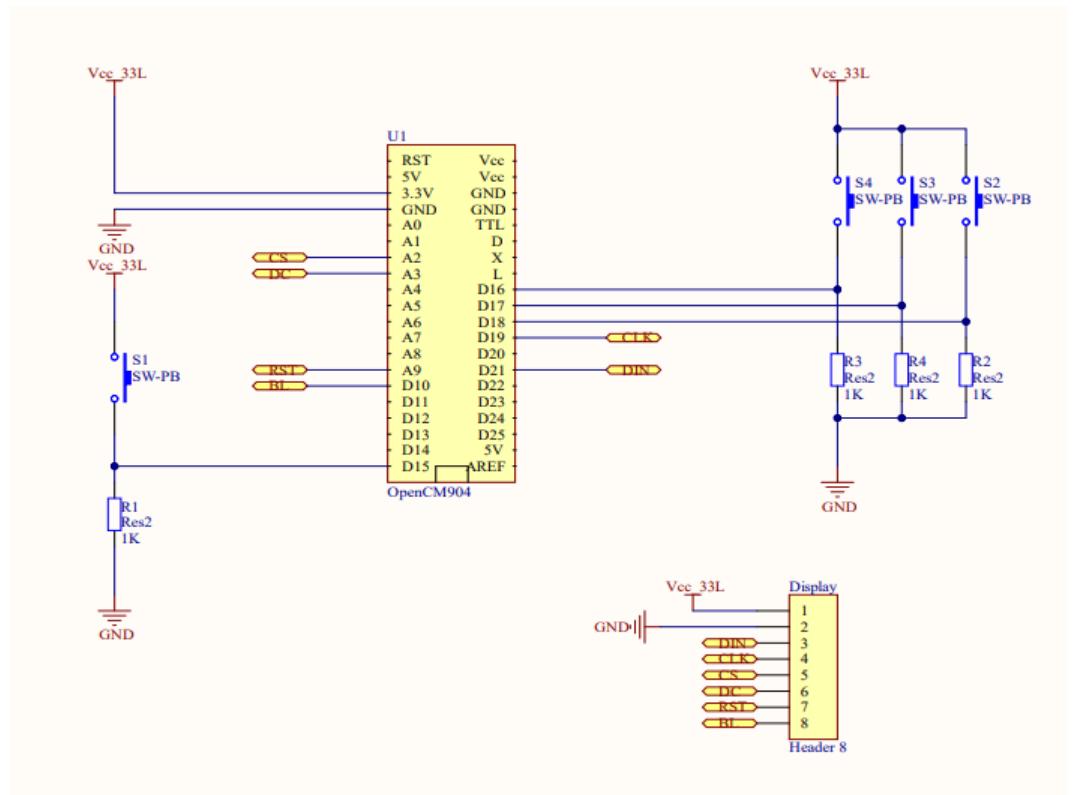


Abbildung 2.1: Schaltplan der Hauptplatine des Roboters

2.2 Komponenten des Roboters

2.2.1 Motoren

Die Servomotoren an dem Roboter sind zwei Motoren der Firma *ROBOTIS* Typ XL330-M077-T. Die Motoren können in verschiedenen Modi betrieben werden:

1. Current Control Mode
2. Velocity Control Mode
3. Velocity Control Mode
4. Current-based Position Control Mode
5. PWM Control Mode

Die zwei Servomotoren werden mit PWM(Pulse-width Modulation) Control Mode angesteuert. Die Motoren werden von dem Mikrocontroller über *TTL Communication* kontrolliert. Darüber hinaus können die aktuelle Geschwindigkeit, aktuelle Drehwinkel, Strom und Eingangsspannung des Motors auch davon ausgelesen werden.



Abbildung 2.2: Servomotor XL330-M077-T von *ROBOTIS Co., Ltd.*

Quelle: XL330-M077-T[Produktbild]. (2023). *ROBOTIS Co., Ltd.* Abgerufen von <https://emanual.robotis.com/docs/en/dxl/x/xl330-m077/>

2.2.2 Monitor

Bei dem Monitor handelt es sich um ein 2.4-Inch-LCD-Modus 1 von *Waveshare Electronics*, das auf dem SPI-Protokoll basiert und hauptsächlich zur Anzeige von Zustand des Roboters verwendet wird.

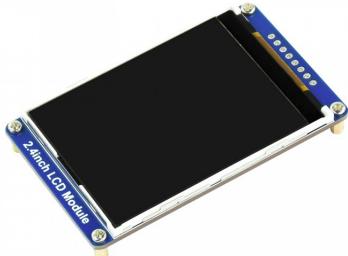


Abbildung 2.3: 2.4-Inch-LCD-Modus 1 *Waveshare Electronics*

Quelle: 240×320, General 2.4inch LCD Monitor Modus 1e, 65K RGB[Produktbild]. (2023). *Waveshare Electronics*. Abgerufen von <https://www.waveshare.com/2.4inch-LCD-Modus-1e.htm>

2.2.3 Mikrocontroller

Als Controller für den Roboter wird OpenCM9.04 Type C der Firma *ROBOTIS* verwendet. Er ist ein ARM-Cortex-M3 Mikroprozessor-basierten Controller mit einer Taktfrequenz von 72 MHz, 128 KB Flash-speicher und 20 KB SRAM-Speicher. Außerdem verfügt er über mehrere gängige I/O-Schnittstellen wie UART, SPI, I2C, PWM usw. Das Hauptmerkmal des OpenCM9.04 im Vergleich zu anderen Controllern ist der direkte Anschluss an den Servomotoren. Dies erlaubt, keine zusätzliche Schaltung für den Antrieb der Motoren zu erforderlich, was den Bedarf an Hardware-Design reduziert. Einzelheiten zu den Software- und Hardwareanwendungen sind unten aufgeführt:

Programmierung

Die Programmierung des Roboters basiert auf dem *Arduino-Framework* [21] und ist in Programmiersprache C und C++ geschrieben. Der Roboter ist sowohl in Modus 1 als auch in Modus 2 mit zwei Schleifen programmiert, wobei die Abtastfrequenz der Schleifen leicht unterschiedlich ist.

In Modus 1 liest Schleife 1 die Batteriespannung mit einer Rate von 1 Hz ab, während Schleife 2 die Hauptarbeit von Modus 1 übernimmt und den Monitor mit einer Rate von 25 Hz aktualisiert.

Im Programm für Modus 2 liest Schleife 1 die Batteriespannung und aktualisiert den Monitor mit 10 Hz, während Schleife 2 die Exploration von Modus 2 mit 25 Hz durchführt. Da Modus 2 in erster Linie auf die Exploration der Umgebung ausgerichtet ist, handelt es sich um eine langfristige Aufgabe, die keine ständige Aktualisierung des Monitors erfordert, so dass die Abtastfrequenz von Schleife 1 reduziert wird, um den Aufwand zu verringern.

Hardware-Interrupts

Auf den vier Tasten des Roboters sind vier Hardware-Interrupts eingerichtet, die grundlegende Vorgänge wie die Unterbrechung des Laufs und die Fortsetzung des Laufs usw. ermöglichen.



Abbildung 2.4: OpenCM9.04 typeC Mikrocontroller

Quelle: OpenCM9.04 typeC[Produktbild]. (2023). ROBOTIS Co., Ltd. Abgerufen von <https://emanual.robotis.com/docs/en/parts/controller/opencm904/>

2.2.4 Mechanische Konfiguration für Modus 1 und Modus 2

In Modus 1 und Modus 2 ist die mechanische Konfiguration unterschiedlich. In Modus 1 sind die beiden Servomotoren jeweils nur mit einer Stange verbunden, in Modus 2 sind die beiden Motoren über ein fünfgliedriges Koppelgetriebe miteinander verbunden. Das Drehgelenk in der Mitte des Mechanismus hat eine Isolationssäule als Endeffektor, der zur Wahrnehmung der Umgebung

dient. Die Abbildungen 2.5 und 2.6 zeigen das 3D-Modell und die Draufsicht der technischen Zeichnung der beiden Modi, in der nur die Längen der Stangen angegeben sind.

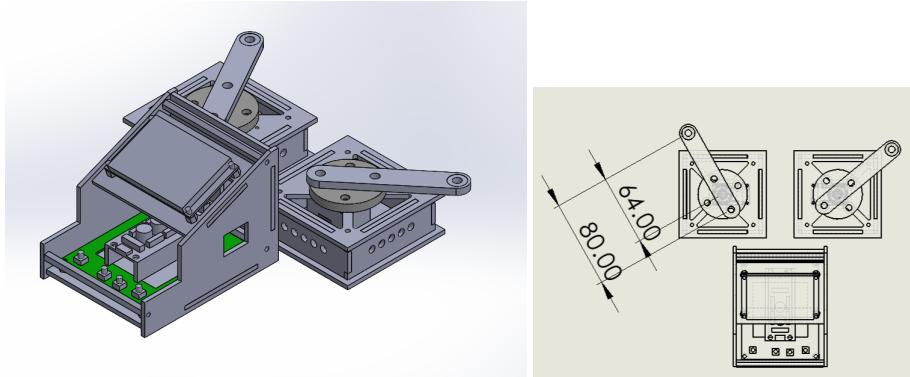


Abbildung 2.5: 3D-Modell und Technische Zeichnung bei Modus 1

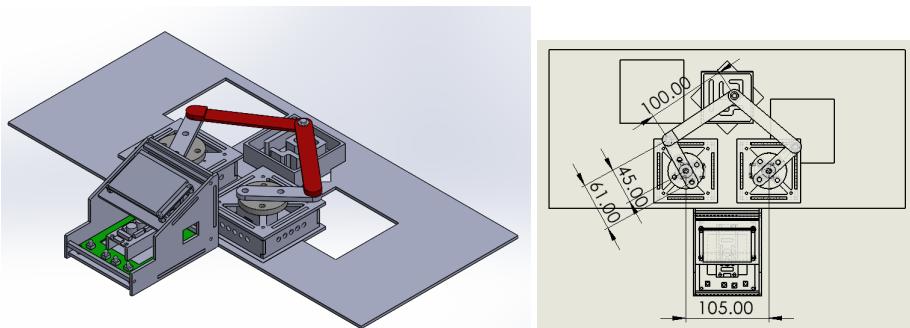


Abbildung 2.6: 3D-Modell und Technische Zeichnung bei Modus 2

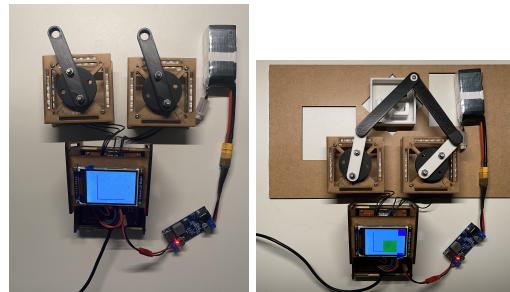


Abbildung 2.7: Aufbau des Roboters bei Modus 1 (links) und Modus 2 (rechts)

Kapitel 3

Thematische Grundlagen und praktische Umsetzungen

Im Folgenden werden die Grundlagen, die sowohl in diesen beiden Modi gemeinsam angewendet werden, als auch in ihren jeweiligen Einzelanwendungen vorgestellt. Die angewandten Grundlagen im Rahmen dieser Arbeit werden sowohl in allgemeiner Form als auch in ihrer praktischen Umsetzung beschrieben.

3.1 Die theoretischen Grundlagen von Modus 1 und Modus 2

3.1.1 Freiheitsgrad

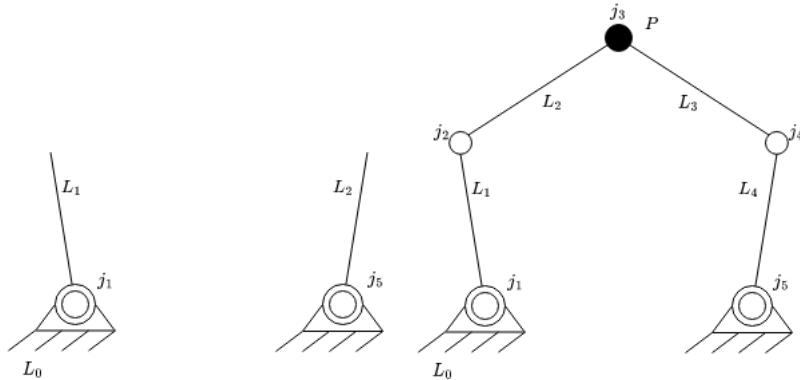


Abbildung 3.1: Schematische Darstellung der Mechanismen bei Modus 1 (links) und Modus 2 (rechts)

Der Freiheitsgrade eines Mechanismus ist definiert als die minimalen unabhängigen Parameter, die erforderlich sind, um die Position jedes Bauteils im Mechanismus zu beschreiben. Um den Freiheitsgrad des Mechanismus des Roboters zu bestimmen, kann die Grüblersche Gleichung wie folgt verwendet werden [17]:

$$F = m(N - 1) - \sum_{i=1}^j c_i \quad (3.1)$$

F : Freiheitsgrad

m : Typ des Getriebes ($m = 6$ für räumliches, $m = 3$ für ebenes Getriebe)

N : Anzahl der Getriebeglieder (L_i)

j : Anzahl der Gelenke (j_i)

c_i : Zwangsläufigkeit eines einzelnen Gelenks

In dieser Arbeit bewegt sich der Roboter mit Drehgelenken ($c_i = 2$) in einer Ebene ($m = 2$). Der Freiheitsgrad des Mechanismus für Modus 1 und Modus 2 kann wie folgt berechnet werden:

Modus 1:

$$\begin{aligned} j &= 2 \\ N &= 3 \\ F &= 3(3 - 1) - \sum_{i=1}^2 2 = 2 \end{aligned}$$

Modus 2:

$$\begin{aligned} j &= 5 \\ N &= 5 \\ F &= 3(5 - 1) - \sum_{i=1}^5 2 = 2 \end{aligned}$$

3.1.2 Configuration

In Robotik ist *Configuration* eine Darstellung, die verwendet wird, um den Zustand eines Roboters vollständig zu repräsentieren. Für einen Roboter kann die *Configuration* durch einen Satz von Koordinaten (mobiler Roboter) oder durch einen Satz von Gelenkwinkeln (Robotermanipulator), abhängig von der Bewegung und *Configuration* des Roboters dargestellt werden. Abgesehen davon wird in Buch "Modern Robotics" [17, s.11] eine gute Erklärung gegeben: "*The minimum number n of real-valued coordinates needed to represent the configuration is the number of degrees of freedom (dof) of the robot.*"

Nach [17] ist die Dimension der *Configuration* der Freiheitsgrad. Es wurde auch bereits in 3.1.1 festgestellt, dass der Freiheitsgrad beider Modi 2 ist, d. h. die Dimension der *Configuration* ist 2. Hier kann *Configuration* des Roboters durch (ϕ_1, ϕ_2) beschrieben werden, wobei $\phi_1 [^\circ]$ und $\phi_2 [^\circ]$ die Drehwinkel der beiden an Drehgelenk 1 und Drehgelenk 5, wie in Abbildung 3.1 gezeigt, sind.

3.1.3 Configuration Space

Das *Configuration Space* (*C-Space*) wird in Robotik zur Beschreibung aller möglichen *Configuration* verwendet. Er ist der **abstrakte mathematische** Raum, in dem alle möglichen Configurationen zusammengefasst sind. Jeder Zustand des Roboters wird durch einen Punkt im *C-Space* dargestellt.

Der Freiheitsgrad ist bekanntlich die Anzahl der unabhängigen Variablen, und der Freiheitsgrad entspricht der Dimension der *Configuration*. Wenn also das System durch N unabhängige Variablen gesteuert wird, ist sein *C Space* ein N -dimensionaler Raum.

In dieser Arbeit werden also zwei Motoren an einen Roboter montiert, dessen Drehwinkel ein unabhängige Variable ist. Der sich daraus ergebende Freiheitsgrad der beiden Modi ist 2, so dass die Dimension von *Configuration* und dem durch die Configuration gebildeten *C Space* ebenfalls 2 ist. Der sich daraus ergebende 2-dimensionale Raum S spielt in dieser Arbeit eine wichtige Rolle.

$$S = \{(\phi_1, \phi_2) \mid \phi_1 \in [0, 360), \phi_2 \in [0, 360)\} \quad (3.2)$$

Kollisionserkennung

Der verwendete Servomotor XL-330-M7 ist ein Gleichstrommotor, der zur Erkennung einer Kollision mit einem Hindernis unter Verwendung von Strom verwendet werden kann. Die Versorgungsspannung des Motors sei U , der Innenwiderstand R , *Back EMF* E und der Strom I . Die Gleichung für die Spannung im stationären Zustand ergibt sich wie folgt[9]:

$$U = E + IR \quad (3.3)$$

Laut [12] *Back EMF* lässt sich von *Back-EMF*-Konstant K_b und Motordrehzahl ω bestimmen.

$$E = K_b\omega \quad (3.4)$$

Das bedeutet, dass E proportional zu ω ist, d. h. unter der Annahme, dass der Motor im Idealfall während die Drehung durch ein Hindernis blockiert wird, bedeutet dies, dass bei konstantem U , $\omega = 0$, $E = 0$ ist, was gemäß der Gleichung einen Anstieg von I zur Folge hat. Hier ist es also möglich, eine Kollision zu erkennen, indem man den Stromwert abliest, um zu sehen, ob ein bestimmter Schwellenwert überschritten wird. Es ist kein zusätzlicher Sensor erforderlich, um dies zu erreichen.

3.2 Die theoretischen Grundlagen von Modus 1

3.2.1 PID-Regler

Ein PID-Regler ist ein Regler mit geschlossenem Regelkreis, der Steuersignale erzeugt, um das Verhalten des Systems durch den Vergleich der Differenz zwischen dem Sollwert und dem Istwert (d. h. der Regelabweichung) anzupassen. Der PID-Regler besteht aus drei Anteilen: P-Anteil, I-Anteil und D-Anteil. Nachfolgend wird ein kurzer Überblick über die drei Anteile gegeben:

1. Die Funktion des P-Anteils besteht darin, die Ausgangsgröße durch Skalierung der Abweichung zwischen dem Sollwert und dem Istwert einzustellen.
2. Der I-Anteil dient zur Anpassung der Ausgangsgröße mithilfe der über die Zeit akkumulierten Abweichung und kann zur Eliminierung von bleibender Regelabweichung eingesetzt werden.
3. Der D-Anteil wird verwendet, um die Ausgangsgröße entsprechend der Änderungsrate der Differenz zwischen der aktuellen und der vergangenen Abweichung einzustellen. Dieser wird oft verwendet, um die Reaktionszeit des Systems zu verbessern.

Der mathematische Ausdruck für den Regler in einem digitalen System lautet wie folgt[1]:

$$u_k = K_p e_k + K_i \sum_{j=0}^k e_j + K_d (e_k - e_{k-1}) \quad (3.5)$$

- k : Abtastfolge
 u_k : Ausgangsgröße zum k -ten Abtastzeitpunkt
 e_k : Abweichung zum k -ten Abtastzeitpunkt
 e_{k-1} : Abweichung zum $k - 1$ -ten Abtastzeitpunkt
 K_p : Propotionalsfaktor
 K_i : Integrationsfaktor
 K_d : Differentialfaktor

Die P-, I- und D-Anteile können durch die Parameter K_p , K_i und K_d in Gleichung 3.5 eingestellt werden. Durch Entfernen des D-Anteils (K_d auf 0 setzen) oder des I-Anteils (K_i auf 0 setzen) ergeben sich drei weitere Varianten von P-Reglern, PI-Reglern und PD-Reglern. In dieser Arbeit wird nur der PD-Regler verwendet, d. h. der I-Anteil wird entfernt.

3.3 Die theoretischen Grundlagen von Modus 2

3.3.1 Work Space

In der Robotik ist der *Work Space* des Roboters der Bereich im **echten** Raum, in dem der Roboter tatsächlich arbeiten und die ihm zugesetzte Aufgabe erfüllen kann. Er wird durch die Grenzen der Bewegung des Roboters definiert. Im Allgemeinen kann der *Work Space* als der Bereich definiert werden, den der Endeffektor erreichen kann. Ein Endeffektor wiederum ist eine Komponente, die am Ende eines Robotermanipulators angebracht ist und zur Interaktion mit der Umgebung dient. Beispiele für Endeffektoren sind Greifer, Saugnäpfe oder Proben.

Der Roboter bei Modus 2 verwendet das fünfgliedrige Koppelgetriebes zur Steuerung des Endeffektors zur Exploration der Umgebung. Zur Bestimmung des Bereichs, den der Roboter erkunden kann, ist es notwendig; *Work Space* des Roboters zu untersuchen.

Das *Work Space* des fünfgliedrigen Koppelgetriebe kann wie [7] gezeigt bestimmt werden:

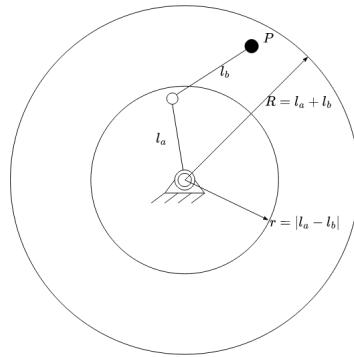
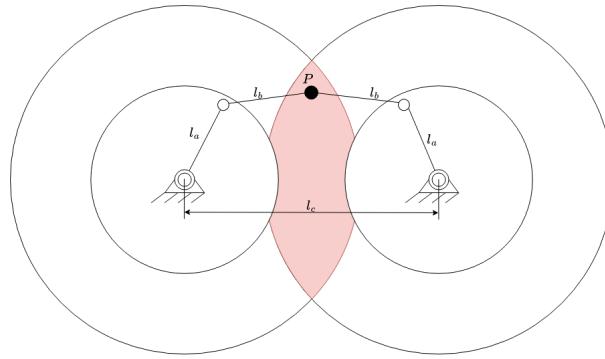


Abbildung 3.2: *Work Space* der zwei-Drehgelenke-in-Reihe

1. Ein fünfgliedriges Koppelgetriebe kann in zwei-Drehgelenke-in-Reihe unterteilt werden. Der *Work Space* des Mechanismus ist der Bereich, den der Punkt P , nämlich der Endeffektor, erreichen kann (siehe Abbildung 3.2).
2. Der *Work Space*, in dem sich die beiden zwei-Drehgelenke-in-Reihe überlappen, ist der *Work Space* des Roboters (siehe Abbildung 3.3).

Abbildung 3.3: *Work Space* des auf fünfgliedrigem Koppelgetriebe basierenden Roboters

Entsprechend der Beziehung zwischen l_a , l_b und l_c in Abbildung 3.3 lässt sich der Roboter, je nach *Work Space*, in 5 Typen unterteilen. Eine ausführliche Aufschlüsselung der allen Konfigurationen ist in Abbildung 3.4 dargestellt.

Type	Shape	Condition
1		$\begin{cases} l_c - r < R < l_c + r \\ r > \frac{l_c}{2} \end{cases} \Leftrightarrow \begin{cases} 2l_a < l_c < 2l_b \\ l_b > \frac{l_c}{2} + l_a \end{cases} (l_b > l_a)$
2		$\begin{cases} l_c - r < R < l_c + r \\ r \leq \frac{l_c}{2} \end{cases} \Leftrightarrow \begin{cases} 2l_a < l_c < 2l_b \\ l_b \leq \frac{l_c}{2} + l_a \end{cases} (l_b < l_a)$
3		$\begin{cases} R \geq l_c + r \\ r > \frac{l_c}{2} \end{cases} \Leftrightarrow \begin{cases} l_c \leq 2l_a \\ l_b > \frac{l_c}{2} + l_a \end{cases} (l_b > l_a)$
4		$\begin{cases} R \geq l_c + r \\ r \leq \frac{l_c}{2} \end{cases} \Leftrightarrow \begin{cases} l_c \leq 2l_a \\ l_b \leq \frac{l_c}{2} + l_a \end{cases} (l_b < l_a)$
5		$\begin{cases} \frac{l_c}{2} \leq R \leq l_c - r \\ r > \frac{l_c}{2} \end{cases} \Leftrightarrow \begin{cases} 2l_b \leq l_c \leq 2l_a + 2l_b \\ l_b > \frac{l_c}{2} + l_a \end{cases} (l_b > l_a)$

Abbildung 3.4: 5 Typen der Konfigurationen eines Roboters mit fünfgliedrigem Koppelgetriebe
Quelle:[7, table. 1]

3.3.2 Singularität

In der Robotik kann eine Singularität intuitiv als eine bestimmte Position im *Work Space* verstanden werden, an der ein Robotermanipulator oder ein kinematisches System gesamt oder eines Teils des Freiheitsgrads verliert. Aus mathematischer Sicht gibt es in den kinematischen Gleichungen eines Robotermanipulators oder Roboters eine Matrix, die sogenannte Jacobi-Matrix, die die Abbildung der *Configuration* aus jeder Robotergelenkposition auf die Änderungsrate der Endeffektor-Haltung im *Work Space* beschreibt. Das Auftreten von Singularitäten führt dazu, dass die Jacobi-Matrix keinen vollen Rang hat, d. h. die transformierten Vektoren sind nicht linear unabhängig, und der Mechanismus des Roboters verliert Freiheitsgrade. Ein Roboter, der sich in der Ebene beliebig bewegen kann, kann sich zum Beispiel an der Singularität nur parallel zur Ein-Achse bewegen.

Die vier Fälle des Auftretens von Singularitäten eines Roboter fünfgliedrigem Koppelgetriebe in dieser Arbeit sind in Abbildung 3.5 dargestellt:

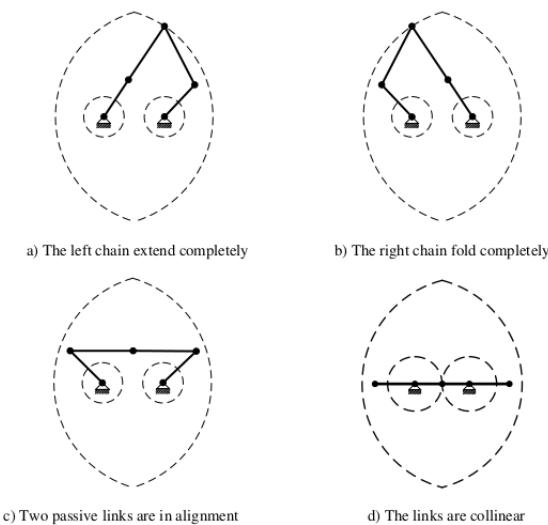


Abbildung 3.5: Singularität des fünfgliedrigen Koppelgetriebes
Quelle:[7, figure. 8]

Konstruktionsbedingt sind Singularitäten des Typs I(a und b in Abbildung 3.5) immer vorhanden und treten an der Grenze des *Work Space* auf. Diese können durch Einschränkung vom *Work Space* vermieden werden, während Singularitäten des Typs II(c in Abbildung 3.5) und III(d in Abbildung 3.5) durch geeignete Werte für die Verbindungslänge beseitigt werden können. Um das Auftreten von Singularitäten zu vermeiden und Interferenzen zwischen zwei aktiven Stangen zu verhindern, wird die Konfiguration vom **Typ 1** (1 in Abbildung 3.4) an Roboter in dieser Arbeit verwendet [7].

3.3.3 Minimal Recurrent Controller

Der Minimal Recurrent Controller(MRC) ist ein neuronales Netz mit Rückkopplung, das aus zwei Neuronen besteht (siehe Abbildung 3.6). Es wird in [11, 19] für die Steuerung eines Differenzialroboters mit zwei Abstandssensoren zum autonomen Explorieren und zur Hindernisvermeidung der Umgebung vorgeschlagen. MRC ist ein *reactive* Ansatz, bei dem der Roboter die Umgebung sehr robust erkunden und Hindernissen ausweichen kann. Im Vergleich zum Braintenberg-Viechle [3] lässt sich damit das Problem vermeiden, dass Roboter, die mit zwei Abstandssensoren ausgestattet sind, häufig in Ecken stecken bleiben. Die Eingangseinheiten dieses neuronalen Netzes sind S_l und S_r , die die Werte der Abstandssensoren zwischen -1 und 1 abbilden (wo -1 für kein Hindernis und 1 für ein erkanntes Hindernis steht); die Ausgangswerte sind für die Rückwärts- bzw. Vorwärtsbewegung M_l und M_r , die ebenfalls zwischen -1 und 1 liegen. Als Aktivierungsfunktion für die Neuronen N_1 und N_2 wird in diesem neuronalen Netz die tanh-Funktion verwendet. Laut [19] können die Beziehungen zwischen den Gewichten folgendermaßen angeordnet werden, um die gewünschte Wirkung dieses neuronalen Netzwerks zu erzielen:

$$w_1, w_3 > 1 \quad (3.6)$$

$$w_r \approx -4w_1, w_l \approx -4w_2 \quad (3.7)$$

$$w_2 \approx -2w_1, w_4 \approx -2w_3 \quad (3.8)$$

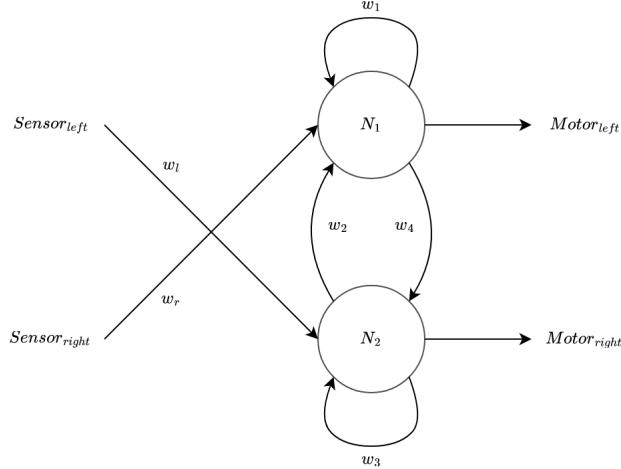


Abbildung 3.6: MRC Neuronales Netzwerk

3.3.4 Moment in Bildverarbeitung

Moment ist eine mathematische Methode zur Beschreibung eines Bildes. Die Pixelwerte und Koordinatenpositionen in einem Bild werden in eine Reihe von mathematischen Werten umgewandelt und bei Anwendungen wie der Bildverarbeitung, Bilderkennung und Bildanalyse eingesetzt. Moment wird häufig zur Beschreibung segmentierter Bildobjekte verwendet. Aus Momenten lassen sich eine Reihe von Eigenschaften eines Bildes, darunter die Fläche, Informationen über das geometrische Zentrum oder die Drehrichtung bestimmen. Die in dieser Arbeit verwendeten Momente sind wie folgt [14, 8]:

- Nicht zentrierte Momente

$$m_{pq} = \sum_x \sum_y x^p y^q I(x, y) \quad (3.9)$$

- Zentrale Momente

$$\mu_{p,q} = \sum_i \sum_j (x - \bar{x})^p (y - \bar{y})^q I(x, y), \bar{x} = \frac{m_{10}}{m_{00}}, \bar{y} = \frac{m_{01}}{m_{00}} \quad (3.10)$$

- Skalierungsinvariante Momente

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{(1+\frac{p+q}{2})}} \quad (3.11)$$

wobei $p + q$ die Ordnung der Momente und x und y die Pixelkoordinaten sind.

- Hu-Momente

$$\begin{aligned} Hu1 &= \eta_{20} + \eta_{02} \\ Hu2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\ Hu3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\ Hu4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\ Hu5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})((\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2) \\ &\quad + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) \\ Hu6 &= (\eta_{20} - \eta_{02})((\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\ Hu7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})((\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2) \\ &\quad - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) \end{aligned}$$

Wobei $I(x, y)$ der Pixelwert in Position (x, y) im Bild ist. In dieser Arbeit werden binäre Bilder verwendet, also ist $I(x, y)$:

$$I(x, y) = \begin{cases} 1 & \text{Objekt} \\ 0 & \text{Hintergrund} \end{cases} \quad (3.12)$$

Der in dieser Arbeit verwendete Deskriptor höchster Ordnung ist $p + q \leq 2$, wobei Deskriptor 0. Ordnung $p + q = 0$ dient in der Regel zur Beschreibung der Fläche, Deskriptor 1. Ordnung $p + q = 1$ zur Beschreibung des Schwerpunkts und Deskriptor 2. Ordnung $p + q = 2$ zur Beschreibung der Verteilung der Pixel verwendet wird [27].

Kapitel 4

Modus 1

In diesem Kapitel werden im Detail der Zweck, die Funktionsweise , die Ergebnisse und eine Zusammenfassung bei Modus 1 beschrieben.

4.1 Verwendungszwecke des Modus 1

bei Modus 1 kann der Benutzer Motor 1 M_1 manuell so drehen, dass Motor 2 M_2 immer im gleichen Winkel bleibt (siehe Abbildung 4.1). Tritt während der Drehung von Motor 2 nach M_1 ein Hindernis (d. h. eine Kollision) auf, so muss M_1 möglichst nahe an der Position bleiben, an der M_2 aufgrund der Kollision nicht mehr rotieren kann. In diesem Fall kann der Benutzer die Stange am M_1 nicht mehr weiterdrehen, aber er kann die virtuelle Kollision durch Berührung wahrnehmen. Darüber hinaus wird die Position, an der die Kollision stattgefunden hat, auf einem externen Monitor visualisiert (siehe Abbildung 4.2).



Abbildung 4.1: Schematische Darstellung des Roboters bei Modus 1

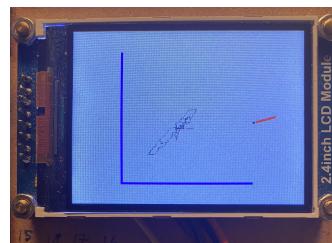


Abbildung 4.2: Der Monitor am Roboter bei Modus 1
Eine Collision bei circa 180°

4.2 Überblick über den Algorithmus

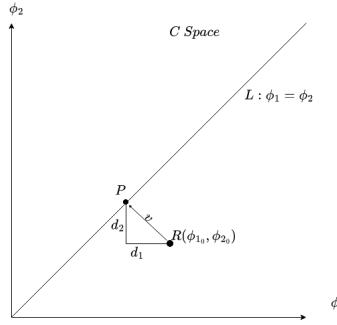


Abbildung 4.3: Schematische Darstellung des *C Space* bei Modus 1

Bei Modus 1 liegt das Problem daran, dass sich beide Motoren immer in der gleichen Position befinden müssen. Im Sinne des *C Space* gilt nämlich, dass $R(\phi_{10}, \phi_{20})$, die aktuelle *Configuration*, bei Modus 1 immer auf der Geraden $L : \phi_1 = \phi_2$ bleiben muss (siehe Abbildung 4.3). Demnach besteht das Hauptproblem darin, die Motoren so zu steuern, dass sich bei einer Abweichung der Punkt R zum Projektionspunkt P bewegen muss, da P der nächstgelegene Punkt zu R auf der Geraden $\phi_1 = \phi_2$ ist.

4.3 Detaillierte Beschreibung des Algorithmus

Im Algorithmus werden hier die beiden PD-Regler zur Steuerung von zwei Motoren verwendet, um die horizontale bzw. vertikale Position von R und die Eingangsgrößen für die beiden Regler zu steuern. Die Abweichung, d. h. die Abweichung zwischen den Positionen auf P und R in horizontaler und vertikaler Richtung, sind d_1 und d_2 in Abbildung 4.3, die wie folgt bestimmt werden können:

Um d_1 und d_2 zu ermitteln, muss zunächst die Koordinate von P bestimmt werden, welche durch die Projektion auf eine Gerade bestimmt werden kann:

$$P\left(\frac{b^2\phi_{10} - ab\phi_{20} - ac}{a^2 + b^2}, \frac{-ab\phi_{10} + a^2\phi_{20} - bc}{a^2 + b^2}\right) \quad (4.1)$$

Dabei sind a, b und c die Koeffizienten der linearen Gleichung $L : a\phi_1 + b\phi_2 + c = 0$, hier $a = 1$, $b = -1$ und $c = 0$. Das Einsetzen von a, b und c in Gleichung 4.1 ergibt P :

$$P\left(\frac{\phi_{10} + \phi_{20}}{2}, \frac{\phi_{10} + \phi_{20}}{2}\right) \quad (4.2)$$

Da in der Regelungstechnik für $Abweichung = Sollwert - Istwert$ gilt, können d_1 und d_2 wie folgt bestimmt werden:

$$d_1 = \frac{\phi_{10} + \phi_{20}}{2} - \phi_{10} = \frac{\phi_{20} - \phi_{10}}{2} \quad (4.3)$$

$$d_2 = \frac{\phi_{10} + \phi_{20}}{2} - \phi_{20} = \frac{\phi_{10} - \phi_{20}}{2} \quad (4.4)$$

Hierbei werden d_1 und d_2 als Eingangsgrößen der PD-Regler verwendet. Eine schematische Darstellung des Regelkreises ist in Abbildung 4.4 dargestellt.

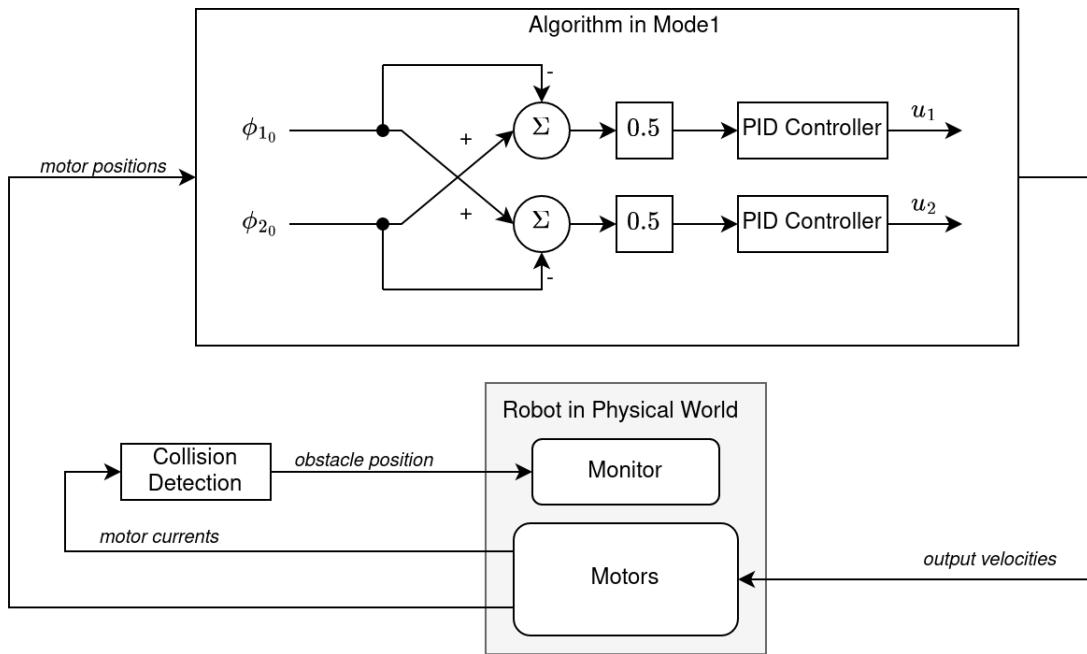


Abbildung 4.4: Schematische Darstellung zum Modus 1 Algorithmus

4.4 Ergebnisse

C Space basierte Positionskontrolle

In den folgenden Abbildungen ist eine visuelle Darstellung des Roboters *C Space* unter Verwendung der PD-Regelung bei Modus 1 zu sehen. Die gelben Punkte beziehen sich auf *Configuration*. Es ist deutlich zu erkennen, dass sich *Configuration* neben oder auf der Linie der Mittellinie $\phi_1 = \phi_2$ befindet. Einige *Configuration* sind aufgrund des Winkelunterschieds zwischen der Stange des Motors leicht versetzt, wenn er vom Benutzer gedreht wird. Wenn die Stange vom Benutzer losgelassen wird, bewegt sich die *Configuration* direkt auf die Mittellinie zu, wie in Abbildung 4.3 dargestellt. Darüber hinaus ist zu erkennen, dass der Proportionalsfaktor K_p die Größe und das Verhalten der mittleren Region beeinflusst, die bei $K_p = 1,0$ am kleinsten ist, im *C Sapce* aber oszilliert. Bei $K_p = 0,3$ ist der mittlere Bereich am größten.

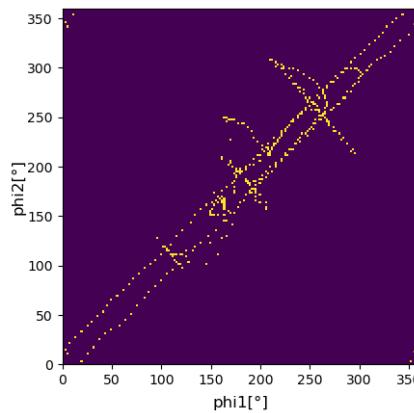


Abbildung 4.5: Visuelle Darstellung des *C Space* bei Modus 1
 PD-Regler, $K_p = 1,0$, $K_d = 0,25$
 Oszillierung bei circa (250, 250); Kollision bei circa (160, 250) und (250, 250)

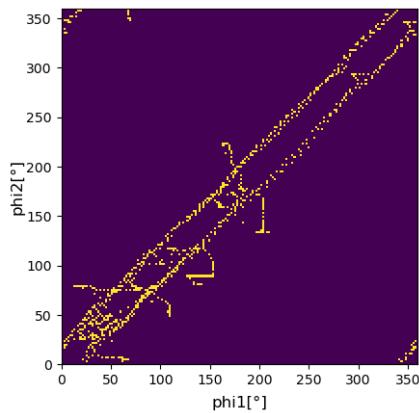


Abbildung 4.6: Visuelle Darstellung des *C Space* bei Modus 1

PD-Regler, $K_p = 0.7, K_d = 0.25$
Kollision bei circa (180, 210) und (200, 140)

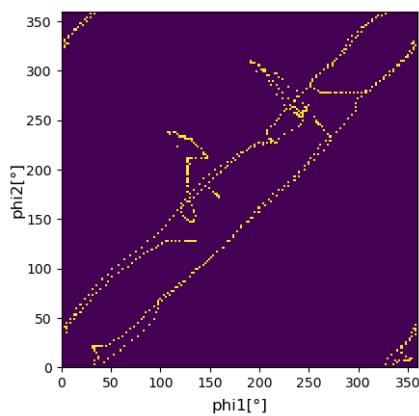


Abbildung 4.7: Visuelle Darstellung des *C Space* bei Modus 1

PD-Regler, $K_p = 0.3, K_d = 0.25$
Kollision bei circa (100, 240) und (140, 170)

Kollisionserkennung

In den Abbildungen 4.5, 4.6 und 4.7 sind Punkte zu sehen, die etwas weiter von der Mittellinie entfernt sind. Diese können als große Winkelverschiebungen aufgrund von Kollisionen angesehen werden, da eine Kollision im Idealfall zu $|\phi_{10} - \phi_{20}| \gg 0$ führen würde. Hier folgt ein weiterer Datensatz, der hervorragend im unteren Diagramm in Abbildung 4.8 zu sehen ist. Die Winkel der beiden Motoren sind die meiste Zeit über gleich, aber die Winkelabweichungen treten bei circa 50s und 70s auf. Das mittlere Diagramm zeigt, ob eine Kollision stattgefunden hat oder nicht (1 als ja, 0 als nein) und das obere Diagramm zeigt, dass die Ströme der beiden Motoren zu diesen beiden Zeitpunkten ansteigen. Das Verhalten von *Configuration* im *C Space* ist entsprechend diesen beiden Punkten im *C Space* außermittig zu beobachten (siehe Abbildung 4.9).

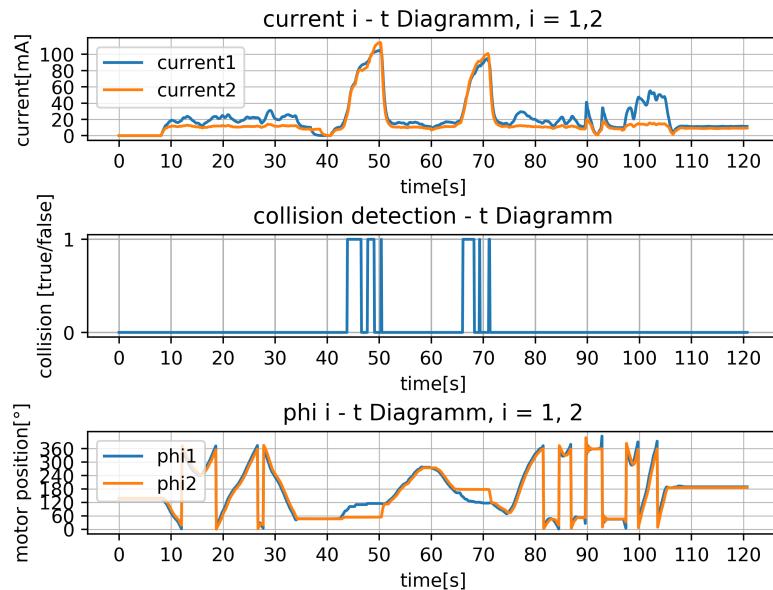


Abbildung 4.8: Strom, Kollisionserkennung und *Configuration* über Zeit-Diagramm
 $current_i$ ist der von Motor i gelesene Strom Kollisionn bei circa 50s und 70s

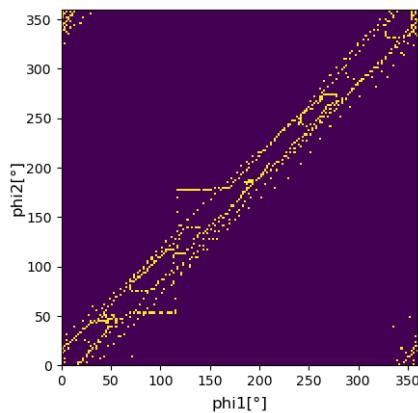


Abbildung 4.9: Visuelle Darstellung des *C Space* bei Modus 1
PD-Regler, $K_p = 0.7$, $K_d = 0.25$
Kollision bei circa (110, 50) und (250, 250)

4.5 Zusammenfassung und Ausblick

Die Ergebnisse von Modus 1 zeigen, dass ein einfaches haptisches Feedback bzw. naive strombasierte Kollisionserkennung realisiert werden kann. Noch wichtiger ist, dass das Verhalten des Roboters einfach durch die Steuerung der *Configuration*, d. h. die Position von R im *C Space* kontrolliert werden kann. Diese beiden Methoden lassen sich in Modus 2 auf den haptischen Explorationsalgorithmus des Roboters erweitern.

Kapitel 5

Modus 2

In diesem Kapitel werden im Detail der Zweck, die Funktionsweise, die Ergebnisse und eine Zusammenfassung bei Modus 2 beschrieben.

5.1 Verwendungszwecke des Modus 2

Für Modus 2 sollte ein modellfreier Explorationsalgorithmus mit sensorlosen haptischem Feedback entwickelt werden. Die Aufgabe in diesem Modus besteht darin, den Roboter in die Lage zu versetzen, zwei 60 mm x 60 mm große 2D-Labyrinthe (rote Quadrate in Abbildung 5.1) durch ein fünfgliedriges Koppelgetriebe zu explorieren (wie in der Abbildung 5.1 gezeigt). Während die Umgebung erkundet wird, werden Karten erstellt. Sobald die beiden Karten erstellt sind, werden sie miteinander verglichen, um festzustellen, ob sie übereinstimmen und wie groß der Unterschied im Drehwinkel ist. Auch hier ist zu beachten, dass das kinematische Modell des fünfgliedrigen Koppelgetriebes dem Roboter unbekannt ist.

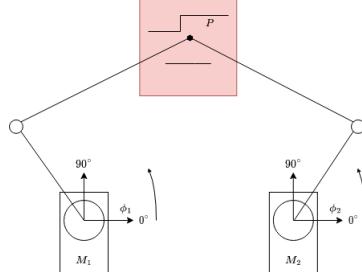


Abbildung 5.1: Schematische Darstellung des Roboters bei Modus 2

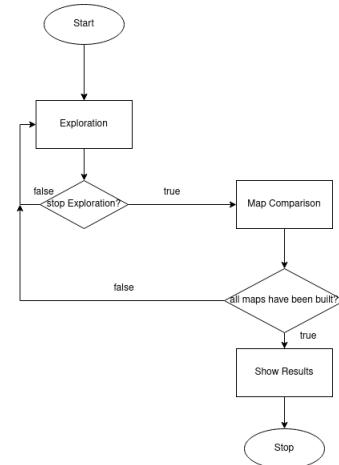


Abbildung 5.2: Prozess bei Modus 2 als Flussdiagramm

5.2 Überblick über den Algorithmus

Der gesamte Prozess besteht aus zwei Teilen, *Exploration* und *Karte Vergleichen* (siehe Abbildung 5.2).

Exploration

Die Exploration basiert auf einem von *MRC* angesteuerten virtuellen Roboter im *C Space* (siehe Abbildung 5.3). Das Grundprinzip des Algorithmus lässt sich wie folgt unterteilen:

1. Ein zweidimensionaler auf den Winkeln der zwei Motoren basierender *C Space* wird erstellt.
2. Ein virtueller mobiler Roboter R mit zwei virtuellen Abstandssensoren im *C Space* wird erstellt; die Positionskoordinaten von R im *C Space* werden entsprechend dem aktuellen Winkel der Motoren definiert.
3. Wenn der Roboter die reale Welt erkundet, wird die auf strombasierte Kollisionserkennung durchgeführt und im Falle einer Kollision wird vor dem virtuellen Roboter im *C Space* ein Hindernis markiert.
4. Während R den *C Space* durch *MRC* untersucht, werden die horizontalen und vertikalen Geschwindigkeiten von R in Bezug auf den *C Space* mithilfe der Kinematik des Differenzialroboters berechnet, die den Motorsteuerungssignalen des realen Roboters entsprechen.
5. Der zugängliche Bereich in ***C Space*** muss vorab durch $\text{Min}_{\phi_1} \leq \phi_1 < \text{Max}_{\phi_1}$ und $\text{Min}_{\phi_2} \leq \phi_2 < \text{Max}_{\phi_2}$ eingeschränkt werden. Der Grund dafür sind die strukturellen Eigenschaften des fünfgliedrigen Koppelgetriebes, bei dem die inverse Kinematik für zwei Lösungen gelöst werden kann, wenn die Position eines Endeffektors gegeben ist. In dieser Arbeit ist die Kinematik jedoch unbekannt, und die mehrfachen Lösungen erhöhen die Komplexität der Karte und können dazu führen, dass der Roboter in eine Singularität eintritt, die ihn daran hindert, richtig zu funktionieren (siehe Abbildung 5.4 und vergleiche mit Abbildung 3.5).

Kartenvergleich

Nach der Erstellung jeder Karte werden bei der Vorverarbeitung die Hindernisse der aktuellen Karte mithilfe des *Connected-Component-Labelling Algorithmus*[6] extrahiert und in einem Feature-Set gespeichert. Nachdem die beiden Karten erstellt wurden, wird das gespeicherte Feature-Set auf der Grundlage der räumlichen Beziehung zwischen den Hindernissen zum Vergleich der beiden Karten verwendet.

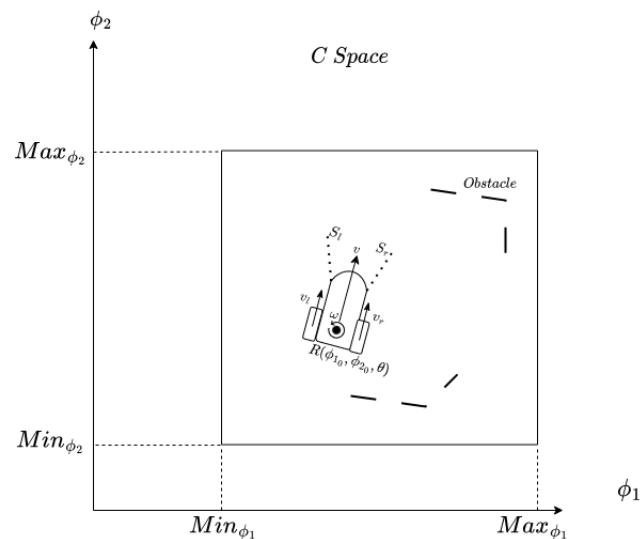


Abbildung 5.3: Virtueller mobiler Roboter R im eingeschränkten C Space

Es sei noch einmal erwähnt, dass ein Differenzialroboter R zwar im Diagramm eingezeichnet ist, aber nicht wirklich existiert. Wie in 3.1.3 erläutert, stellt dieser Punkt R eine *Configuration* dar, und der Parameter θ wird hinzugefügt, um die Richtung des Differenzialroboters zu beschreiben.

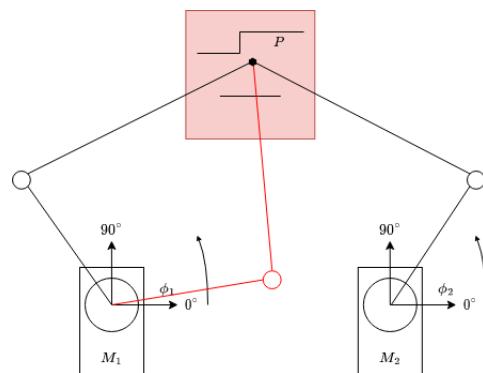


Abbildung 5.4: Der Fall von Mehrfachlösungen eines fünfgliedrigen Koppelgetriebes

5.3 Detaillierte Beschreibung des Algorithmus

Ein Übersicht über den Algorithmus ist in folgender Abbildung gezeigt und die jeweilige Funktionen werden im Detail dargestellt.

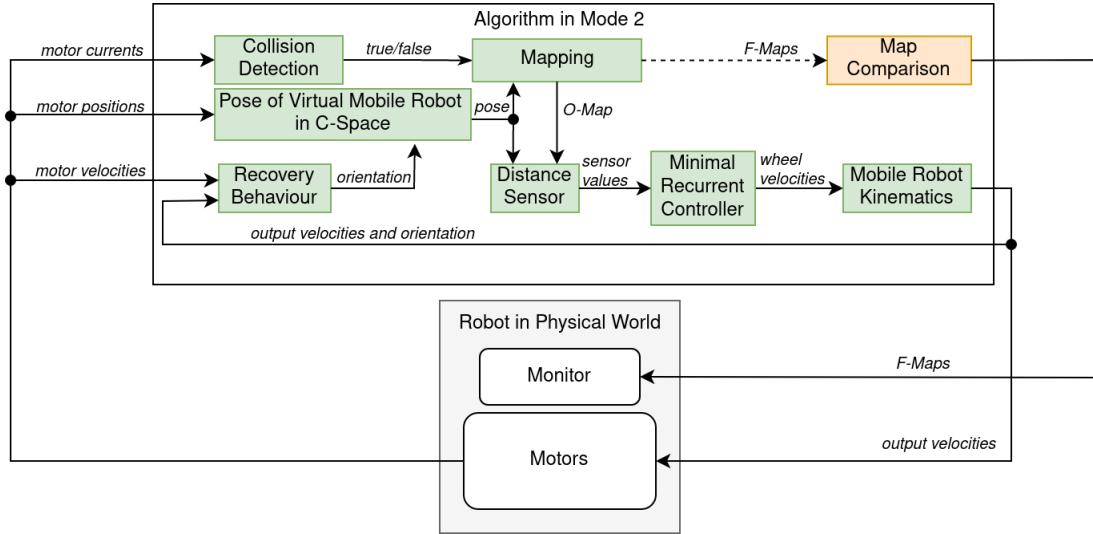


Abbildung 5.5: Übersicht über den MRC-basierten haptischen Explorationsalgorithmus

5.3.1 Exploration

Mapping

Um ein besseres Verständnis der zu untersuchenden Umgebung zu erhalten und das Verhalten des Roboters zu steuern, werden Karten der zu explorierenden Umgebung erstellt. Hier werden zwei binäre Karten, O_{map} und F_{map} , verwendet. Die Position des Hindernisses wird im ersten gespeichert und zur Steuerung des Verhaltens des Roboters verwendet, während im Letzten die vom Roboter erreichte Position gespeichert wird und zur Analyse der Umgebung verwendet werden kann. Die Karten werden im Einzelnen wie folgt erklärt:

- O-Map:** Alle durch Kollisionserkennung markierten Hindernispositionen werden in O_{Map} gespeichert und dem Abstandssensor zur Verfügung gestellt, um die umliegenden Hindernisse zu bestimmen (0 ist leer, 1 ist besetzt). Wie bereits erwähnt, werden zur Einschränkung des zugänglichen Bereichs im C Space, ebenfalls Punkte auf den Linien $Min_{\phi_1} = \phi_1$, $Max_{\phi_1} = \phi_1$, $Min_{\phi_2} = \phi_2$ und $Max_{\phi_2} = \phi_2$ als Hindernisse markiert (siehe Abbildung 5.3).
- F-Map:** Alle Positionen, die von R besucht werden, werden in F_{Map} gespeichert (0 als nicht besucht, 1 als besucht), diese Positionen sind die Drehwinkel der Motoren.

Collision Detection und Obstacle Labeling

Um das Auftreten einer Kollision zu erkennen, kann als naiver Ansatz der Strom als Kriterium verwendet werden: d. h. wenn der Strom von Motoren i_{motor} größer als der Schwellenwert i_{crit} ist, wird davon ausgegangen, dass eine Kollision stattgefunden hat. Bei einer Kollision wird dann ein w breites Raster Hindernis (Wand) senkrecht zum R_v in w Raster vor dem R_v markiert (**nur bei** O_{map}). Die Position der Wand basiert auf dem w Raster vor der aktuellen Roboterposition (pos_x, pos_y) und die Wand wird symmetrisch mit w Punkten entlang $\theta \pm \pi/2$ auf der O_{map} entsprechend der aktuellen Ausrichtung θ markiert. Dies kann wie folgt dargestellt werden:

wenn $i_{motor} > i_{crit}$,

$$x_w = pos_x + w \cos(\theta) \quad (5.1)$$

$$y_w = pos_y + w \sin(\theta) \quad (5.2)$$

$$x_{map} = \lfloor x_w + j \cos(\theta \pm \frac{\pi}{2}) \rfloor, -\frac{w}{2} \leq j \leq \frac{w}{2} \quad (5.3)$$

$$y_{map} = \lfloor y_w + j \sin(\theta \pm \frac{\pi}{2}) \rfloor, -\frac{w}{2} \leq j \leq \frac{w}{2} \quad (5.4)$$

$$O_{map}(x_{map}, y_{map}) = 1 \quad (5.5)$$

Hierbei ist i_{motor} der ausgelesene Strom von Motoren. i_{crit} ist der Schwellenwert für Kollisionserkennung.

Distance Sensor

Mit dieser Funktion werden die Sensorwerte des virtuellen Roboters R berechnet. Zur Vereinfachung werden die Koordinaten des Roboters nun von (ϕ_1, ϕ_2) in (x, y) umgewandelt. Angenommen, der Roboter befindet sich in einem ebenen Koordinatensystem (x, y) , seine Position ist (pos_x, pos_y) und seine Orientierung ist θ . Der Roboter verfügt über vier Sensoren, die mit α_i Grad, $i = 1, 2, 3, 4$ zur linken(1 und 2) und rechten(3 und 4) Seite des Roboters hin abtasten, mit einem maximalen Abstand von d_{max} . Die von den Sensoren S_l und S_r erfasste Entfernung lässt sich mit den folgenden Gleichungen berechnen:

$$x_{avail}^i = pos_x + d_{max} \cdot \cos(\theta + \alpha_i) \quad (5.6)$$

$$y_{avail}^i = pos_y + d_{max} \cdot \sin(\theta + \alpha_i) \quad (5.7)$$

$$x_{sens}(k)^i = \lfloor pos_x(1 - k) + x_{avail}^i \cdot k \rfloor, 0 \leq k \leq 1 \quad (5.8)$$

$$y_{sens}(k)^i = \lfloor pos_y(1 - k) + y_{avail}^i \cdot k \rfloor, 0 \leq k \leq 1 \quad (5.9)$$

für $u = \frac{j}{d_{max}}, 0 \leq j \leq d_{max}$

$$d_i = \begin{cases} \min \sqrt{(pos_x - x_{sens}^i(u))^2 + (pos_y - y_{sens}^i(u))^2} & \text{wenn } O_{map}(x_{sens}^i(u), y_{sens}^i(u)) = 1 \\ max_dis & \text{sonst} \end{cases} \quad (5.10)$$

der Abstand wird dann mit der tanh-Funktion zwischen -1 und 1 abgebildet

$$S_l = \tanh(\frac{d_{max}}{2} - \frac{d_1 + d_2}{2}) \quad (5.11)$$

$$S_r = \tanh(\frac{d_{max}}{2} - \frac{d_3 + d_4}{2}) \quad (5.12)$$

Minimal Recurrent Controller

MRC ist zentraler Bestandteil des haptischen Explorationsalgorithmus. Die Eingangsgrößen des Controllers sind die virtuellen Sensorwerte S_l und S_r von *Distance Sensor* links und rechts, und die Ausgabeeinheit ist die nicht skalierten Radgeschwindigkeit von R links und rechts. Die mathematische Darstellung dieses neuronalen Netzes lässt sich durch die Gleichungen 5.13 und 5.14 ausdrücken.

$$O_l = \tanh(w_r S_r + w_1 O_l + w_2 O_r) \quad (5.13)$$

$$O_r = \tanh(w_l S_l + w_3 O_r + w_4 O_l) \quad (5.14)$$

Mobile Robot Kinematics

Die Kinematik von R ist die Vorwärtsskinematik eines Roboters mit Differenzialantrieb. Nachdem die nicht skalierten Geschwindigkeiten der Räder von R aus MRC gegeben und durch den Skalierungsfaktor s angepasst sind, können die Linear- und Winkelgeschwindigkeiten von R in

Bezug auf den *C Space* berechnet werden. Seine Geschwindigkeit in beiden Achsen kann aus den trigonometrischen Funktionen berechnet werden. Dies wird als PWM-Signal zur Steuerung der Servomotoren verwendet. Die gesamte Kinematik sieht folgendermaßen aus:

$$v_l = sO_l \quad (5.15)$$

$$v_r = sO_r \quad (5.16)$$

$$v = \frac{(v_r + v_l)}{2} \quad (5.17)$$

$$\omega = \frac{(v_r - v_l)}{2b} \quad (5.18)$$

$$\theta = \theta_{t-1} + \omega dt \quad (5.19)$$

$$v_x = v \cos(\theta) \quad (5.20)$$

$$v_y = v \sin(\theta) \quad (5.21)$$

v : Geschwindigkeit des Referenzpunktes R

v_r : Geschwindigkeit des rechten Rades

v_l : Geschwindigkeit des linken Rades

b : Abstand zwischen den Mittelpunkten der beiden Räder

ω : Winkelgeschwindigkeit des Referenzpunktes R

θ : Orientierung des Roboters im *C Space*

v_x : Geschwindigkeit des Roboters auf der horizontalen Achse im *C Space*

v_y : Geschwindigkeit des Roboters auf der vertikalen Achse im *C Space*

Recovery Behaviour

In der Robotik bezieht sich *Recovery Behaviour* auf die Fähigkeit eines Roboters, sich von einem unvorhersehbaren Fehler oder Kontrollverlust durch Selbstkorrektur, Neupositionierung, Neustart oder andere Mittel zu erholen. In vielen praktischen Anwendungen wird die In-situ-Drehung als ein einfaches Wiederherstellungsverhalten für Navigationsroboter angesehen[30, 18]. Dies wird hier auch als *Recovery Behaviour* des R verwendet. Die Auslösebedingung ist gegeben, wenn die Ausgangsgeschwindigkeit der Steuerung deutlich größer ist als die vom Motor zurückgegebene Geschwindigkeit.

5.3.2 Kartenvergleich

Der gesamte Prozess gliedert sich in zwei Teile: **Vorverarbeitung** und **Vergleichen**. Die beiden Karten werden hier als Bilder verarbeitet. **Vorverarbeitung** wird nach Erstellung einer Karte durchgeführt. Die beiden Karten werden dann in **Vergleichen** in ein Feature-Set umgewandelt und miteinander verglichen.

Vorverarbeitung

Nach der Erstellung einer Karte $K_i, i = 1, 2$ wird die Vorverarbeitung wie folgt durchgeführt:

- Filterung und Hintergrundentfernung
Ein Schwellenwertfilter wird angewendet, und der Hintergrund wird aus dem Eingangsbild entfernt. Dann wird das Bild auf eine niedrigere Auflösung heruntergetastet.
- Bildsegmentierung durch Connected Component Labeling (CCL)
Das heruntergetastete Bild wird mit CCL markiert, wobei jedem Pixel ein Label basierend auf der Konnektivität benachbarter Pixel zugewiesen wird. Alle Pixel mit dem gleichen Label werden als ein Feature behandelt.

- Erstellung des Feature-Sets mit Feature-Descriptor

Basierend auf dem durch CCL markierten Bild werden die Features extrahiert und Deskriptoren \vec{D} mit den folgenden Elementen gebildet: Umfang p , Länge l , Breite w , Fläche a und Hu-Moment erster und zweiter Ordnung ($Hu1$ und $Hu2$). Anschließend werden die Deskriptoren des Features dem Feature-Set F_i hinzugefügt und Features mit der Fläche, die kleiner als ein Schwellenwert sind, entfernt. Die Deskriptoren, Features und Feature-Sets können wie folgt dargestellt werden:

$$\vec{D}_{ij} = [p_{ij}, l_{ij}, \dots, Hu1_{ij}, Hu2_{ij}]^T, j = 1 \dots n \quad (5.22)$$

$$F_i = \begin{bmatrix} \vec{D}_{i1} & \vec{D}_{i2} & \dots & \vec{D}_{ij} \\ \bar{x}_{i1} & \bar{x}_{i2} & \dots & \bar{x}_{ij} \\ \bar{y}_{i1} & \bar{y}_{i2} & \dots & \bar{y}_{ij} \\ id_{i1} & id_{i2} & \dots & id_{ij} \end{bmatrix}, j = 1 \dots n \quad (5.23)$$

$$F_i = [\vec{f}_{i1} \quad \vec{f}_{i2} \quad \dots \quad \vec{f}_{ij}], j = 1 \dots n \quad (5.24)$$

n : Maximum für die Anzahl der markierbaren Merkmale

i : Index für i -ten Karte

j : Index für j -ten Feature

f_{ij} : j -ten Feature in i -ten Karte

$(\bar{x}_{ij}, \bar{y}_{ij})$: Schwerpunktsposition des Features f_{ij}

id_{ij} wird ist die Identität des f_{ij} , es wird dann in zweiten Process **Vergleichen** erwähnt.

Vergleichen

Nachdem beide Karten erstellt wurden und der Feature-Satz vervollständigt wurde, wird Folgendes durchgeführt:

- Identitätszuweisung

Jedem f_{ij} wird eine Identität id zugewiesen, die auf dem euklidischen Abstand $d_{Deskrip}(D_{ij}, D_{gh})$ zwischen ihm und dem Deskriptor \vec{D} aller anderen Feature f_{gh} basiert. Das heißt, für f_{ij} findet man ein f_{gh} , $h \neq j$, mit Minimum $d_{Deskrip}$, und wenn $d_{Deskrip}$ kleiner als ein Schwellenwert ist, weist man f_{ij} und f_{gh} die gleiche id , andernfalls eine neue id zu.

$$d_{Deskrip}(f_{ij}, f_{gh}) = \sqrt{(p_{ij} - p_{gh})^2 + \dots + (Hu2_{ij} - Hu2_{gh})^2}, g = 1, 2, h = 1 \dots n, h \neq j \quad (5.25)$$

- Bestimmung der Winkelabweichung durch Summe der euklidischen Abstände

In F_1 werden alle f_{1j} in Position $\bar{x}_{1j}, \bar{y}_{1j}$ schrittweise um 360 Grad(2π) gedreht. Dabei wird der minimale euklidische Abstand zwischen f_{1j} und dem „nächstgelegenen“ Feature f_{2j} im f_2 mit „derselben Identität“ ($id_{1j} = id_{2j}$), berechnet. Diese Abstände von allen f_{1j} werden summiert. Der Winkel mit der kleinsten Summe wird durch Rotation R herausgenommen, um die wahrscheinlichste Winkelabweichung zwischen den beiden Bildern zu bestimmen. Der Prozess kann wie folgt dargestellt werden:

für $\theta \in 0, 2\pi$:

$$[\bar{x}'_{1j}, \bar{y}'_{1j}]^T = R(\theta)[\bar{x}_{1j}, \bar{y}_{1j}] \quad (5.26)$$

$$f_{dis}(\bar{x}_{1j}, \bar{y}_{1j}) = \min(\sqrt{(\bar{x}'_{1j} - \bar{x}_{2h})^2 + (\bar{y}'_{1j} - \bar{y}_{2h})^2}), h = 1 \dots n \quad (5.27)$$

$$\sum_j f_{dis}(\bar{x}'_{1j}, \bar{y}'_{1j}) \quad (5.28)$$

Die wahrscheinlichste Winkelabweichung θ_{min} ist der Winkel, dass die Gleichung 5.28 minimal ist.

- MAPE-basierte Unterscheidung von Karten durch Feature-Positionen und Flächen
Der MAPE (Mean Absolute Percentage Error), der basierend auf den Positionen und Flächen der f_{ij} in den jeweiligen F_i berechnet wurde, wird verwendet, um die Unterschiede zwischen den Karten zu bestimmen. Der MAPE lässt sich wie folgt ableiten:

$$ed_1 = \sum_j \sqrt{x_{1j}^2 + y_{1j}^2} \quad (5.29)$$

$$ed_2 = \sum_j \sqrt{x_{2j}^2 + y_{2j}^2} \quad (5.30)$$

$$MAPE(ed1, ed2) = \frac{|ed1 - ed2|}{\frac{ed1 + ed2}{2}} \quad (5.31)$$

$$A_1 = \sum_j \vec{D}_{1j}[4] = \sum_j a_{1j} \quad (5.32)$$

$$A_2 = \sum_j \vec{D}_{2j}[4] = \sum_j a_{2j} \quad (5.33)$$

$$MAPE(A_1, A_2) = \frac{|A_1 - A_2|}{\frac{A_1 + A_2}{2}} \quad (5.34)$$

5.4 Ergebnisse

In diesem Abschnitt werden die mit dem Explorationsalgorithmus erstellten Karten gezeigt, wiederum in zwei Teilen, **Exploration** und **Kartenvergleich**. Es ist zu erwähnen, dass der Zeitaufwand für die Erstellung jeder vollständigen Karte 15 Minuten beträgt und der zugängliche Bereich im *C Space* wie folgt definiert ist:

$$Min_{\phi_1} = 73[\circ] \quad (5.35)$$

$$Max_{\phi_1} = 184[\circ] \quad (5.36)$$

$$Min_{\phi_2} = 106[\circ] \quad (5.37)$$

$$Max_{\phi_2} = -5(355)[\circ] \quad (5.38)$$

$$(5.39)$$

5.4.1 Exploration

In den Abbildungen 5.6 und 5.7 sind die beiden Labyrinthe in der Draufsicht, die O_{map} (links) und F_{map} (rechts) als Beispiele für Ergebnisse gezeigt. Die Karte in beiden Abbildungen ist das gleiche Labyrinth, aber eines ist gedreht. Betrachtet man das reale Labyrinth und die F_{map} , so kann man feststellen, dass die grundlegenden Merkmalen (die lila Zone im gelben Bereich) der Umgebung gut ausgebildet sind, obwohl die nach der Exploration erstellte F_{map} etwas deformiert ist.

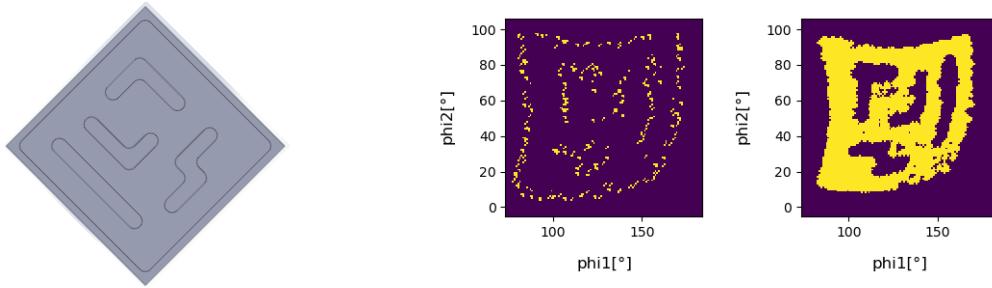


Abbildung 5.6: Ergebnisse der Exploration

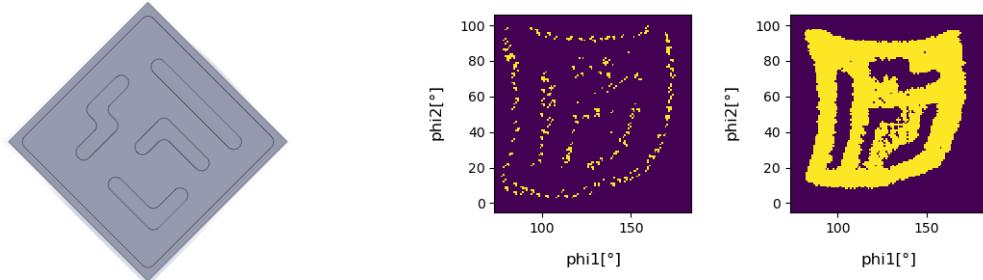


Abbildung 5.7: Ergebnisse der Exploration

Abbildung zwischen Work Space Rahmen und C Space Rahmen

Es ist in den Abbildungen 5.6 und 5.7 deutlich zu sehen, dass die Positionen der Hindernisse bzw. Merkmale in den echten Labyrinthen und in Karte F_{map} nicht exakt übereinstimmen, und F_{map} ist zusätzlichen Drehungen usw. unterworfen. Der Grund dafür ist in Abbildung 5.8 dargestellt, wobei die Bewegung von R entlang der Achse C_{ϕ_1} und C_{ϕ_2} führen zu entsprechenden Bewegungen des Endeffektors P in Richtung W_{ϕ_1} und W_{ϕ_2} , die durch die Rotation der Motoren verursacht werden: Wie auf der linken Seite von Abbildung 5.8 gezeigt, führt eine Bewegung von C_{ϕ_1} zu einer Rechtsdrehung von M_1 und C_{ϕ_2} nach oben zu einer Linksdrehung von M_2 . Durch Betrachtung der Beziehung zwischen C_{ϕ_i} und W_{ϕ_i} , $i = 1, 2$, wie auf der rechten unteren Seite von Abbildung 5.8 gezeigt, kann geschlussfolgert werden, dass W_{ϕ_i} durch horizontale Spiegelung entlang nach links, einer anschließenden Drehung um etwa 45 Grad und einer geringfügigen Streckung in C_{ϕ_i} abgebildet wird. Diese Abbildungsbeziehung wird deutlicher, wenn man sich Abbildung 5.6 und 5.7 ansieht.

5.4.2 Kartenvergleich

Hier werden drei Kartensätze nach dem Heruntertasten und CCL sowie die Ergebnisse des Vergleichens dargestellt. Eine kurze Übersicht der drei Datensätzen folgt:

1. Gruppe 1: Bei dem zu erkundenden Labyrinth handelt es sich um zwei identische Labyrinthe, jedoch mit Winkelabweichungen. Die verwendeten Deskriptoren sind Fläche, Umfang, Länge und Breite.
2. Gruppe 2: Bei dem zu erkundenden Labyrinth handelt es sich um zwei identische Labyrinthe, jedoch mit Winkelabweichungen. Die verwendeten Deskriptoren sind Fläche, Umfang, Länge, Breite sowie Hu-Moment erster und zweiter Ordnung.
3. Gruppe 3: Bei den zu untersuchenden Labyrinthen handelt es sich um zwei verschiedene Labyrinthe mit Winkelabweichungen. Die verwendeten Deskriptoren sind Fläche, Umfang, Länge, Breite sowie Hu-Moment erster und zweiter Ordnung.

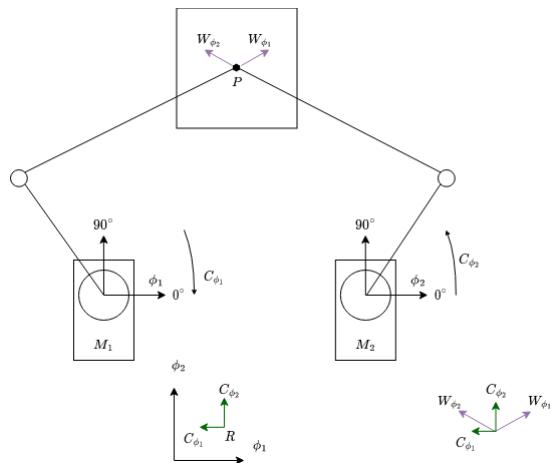


Abbildung 5.8: Schematische Darstellung der Beziehung zwischen *Work Space* und *C Space* des Roboters

Jeder Gruppe werden die Daten in der folgenden Reihenfolge präsentieren:

1. Draufsicht des 3D-Modells der Umgebung.
2. Die heruntergetasteten und durch CCL markierten Karten.
3. Tabellarische Darstellung der entsprechenden Feature-Sets F_1 und F_2 .
4. Normalisierte und nicht-normalisierte Summe der euklidischen Abstände mit unterschiedlichen Drehwinkeln.
5. $MAPE(A_1, A_2)$ und $APE(A_1, A_2)$ werden in Form einer Tabelle dargestellt.

Gruppe 1



Abbildung 5.9: Die Labyrinthe in der echten Welt in Gruppe 1

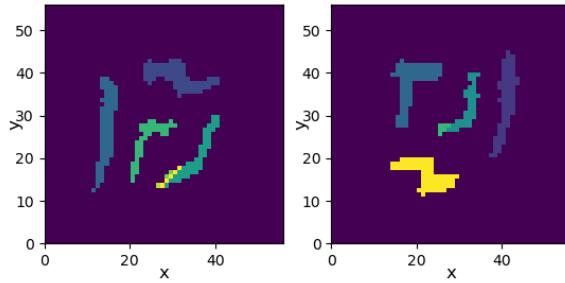


Abbildung 5.10: Die heruntergetasteten und markierten Karten in Gruppe 1 (links Karte 1 K_1 , rechts Karte 2 K_2)

Die Winkelabweichung zwischen zwei Karten ist -180° (180°)

	Feature 1	Feature 2	Feature 3	Feature 4
perimeter	0.4444444	1.0000000	0.5000000	0.1111111
length	0.2294403	1.0000000	0.4905069	0.2329356
width	0.5096185	0.0000000	0.0971618	0.4789809
area	0.9722222	0.9722222	0.2222222	0.0833333
position x	30	13	35	23
position y	38	26	20	23
id	0	1	2	3

Tabelle 5.1: Feature Set von Karte 1 in Gruppe 1

	Feature 1	Feature 2	Feature 3	Feature 4
perimeter	0.9444444	0.6666667	0.0000000	0.3333333
length	0.8510334	0.2089231	0.0875532	0.0000000
width	0.0293110	1.0000000	0.1635907	0.5756669
area	0.8611111	1.0000000	0.0000000	0.6944444
position x	40	18	31	21
position y	32	36	31	15
id	1	4	3	0

Tabelle 5.2: Feature Set von Karte 2 in Gruppe 1

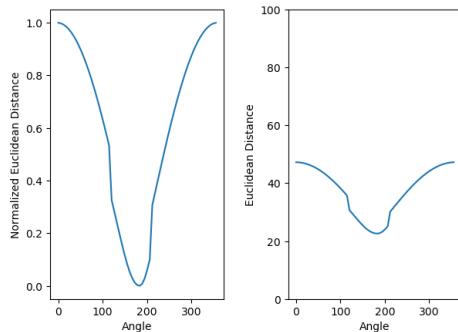


Abbildung 5.11: Die normalisierte und nicht-normalisierte Summe des minimalen euklidischen Abstands über Rotation von 0° bis 360°

θ_{min} liegt bei 183°

Gruppe 2

Abbildung 5.12: Die Labyrinthe in der echten Welt in Gruppe 2

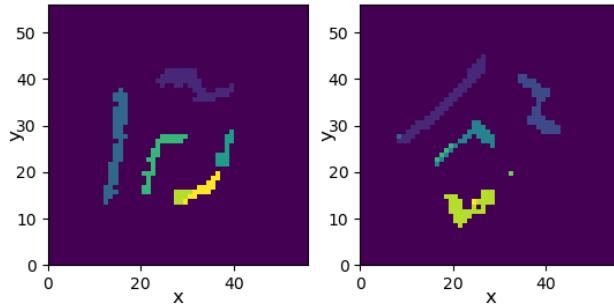


Abbildung 5.13: Die heruntergetasteten und markierten Karten in Gruppe 2(links Karte 1 K_1 , rechts Karte 2 K_2)

Die Winkelabweichung zwischen zwei Karten ist -45° (315°)

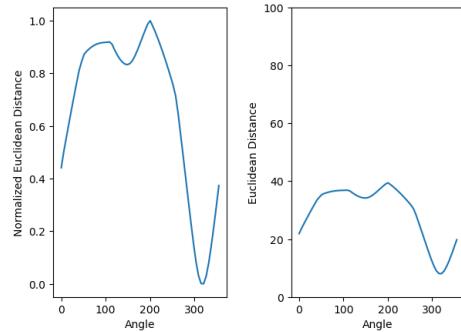


Abbildung 5.14: Die normalisierte und nicht-normalisierte Summe des minimalen euklidischen Abstands über Rotation von 0° bis 360°

θ_{min} liegt bei 320°

	Feature 1	Feature 2	Feature 3	Feature 4
Umfang	0.5151515	0.7575758	0.4545455	0.0909091
Länge	0.3083044	0.8745526	0.3959768	0.1864161
Breite	0.8514799	0.0422882	0.7144658	0.0411113
Fläche	0.9677419	0.9032258	0.4516129	0.0000000
<i>Hu2</i>	0.0880229	0.8638422	0.4621422	0.4460578
<i>Hu2</i>	0.0702350	0.8140637	0.2897522	0.3402381
Position x	30.0000000	14.0000000	23.0000000	36.0000000
Position y	37.0000000	26.0000000	22.0000000	21.0000000
id	0	1	0	2

Tabelle 5.3: Feature Set von Karte 1 in Gruppe 2

	Feature 1	Feature 2	Feature 3	Feature 4
Umfang	1.0000000	0.2727273	0.0000000	0.1515152
Länge	1.0000000	0.2340754	0.1243146	0.0000000
Breite	0.0000000	0.7913867	0.8649035	1.0000000
Fläche	1.0000000	0.5806452	0.0322581	0.3548387
<i>Hu1</i>	1.0000000	0.1553764	0.3893801	0.0000000
<i>Hu2</i>	1.0000000	0.0919608	0.1584698	0.0000000
Position x	18.0000000	38.0000000	23.0000000	22.0000000
Position y	34.0000000	33.0000000	26.0000000	11.0000000
id	1	0	0	0

Tabelle 5.4: Feature Set von Karte 2 in Gruppe 2

Gruppe 3

Abbildung 5.15: Die Labyrinthe in echter Welt in Gruppe 3

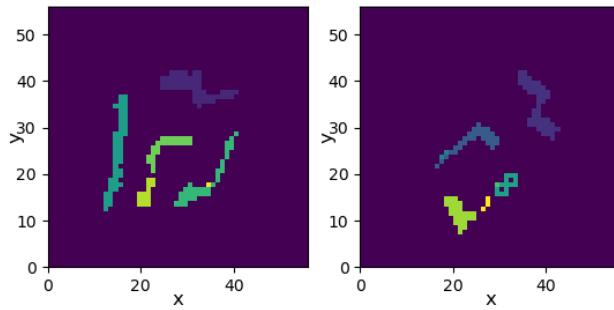


Abbildung 5.16: Die heruntergetasteten und markierten Karten in Gruppe 3(links Karte 1 K_1 , rechts Karte 2 K_2)

Die Winkelabweichung zwischen zwei Karte ist -45° (315°)

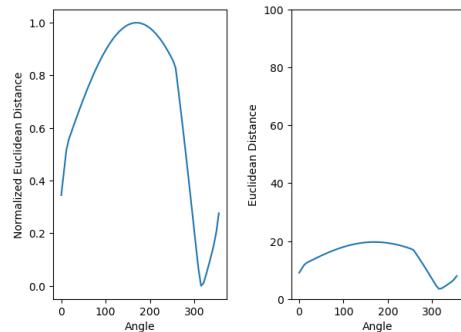


Abbildung 5.17: Die normalisierte und nicht-normalisierte Summe des minimalen euklidischen Abstands über Rotation von 0° bis 360°

θ_{min} liegt bei 315°

	Feature 1	Feature 2	Feature 3	Feature 4
Umfang	0.6969697	1.0000000	0.5151515	0.2727273
Länge	0.4278322	1.0000000	0.7303029	0.2305210
Breite	1.0000000	0.0000000	0.7327873	0.6782928
Fläche	1.0000000	0.8666667	0.3666667	0.0000000
<i>Hu1</i>	0.1903882	1.0000000	0.9588710	0.3450914
<i>Hu2</i>	0.0750356	1.0000000	0.8008420	0.1311741
Position x	30.0000000	14.0000000	33.0000000	24.0000000
Position y	38.0000000	24.0000000	18.0000000	25.0000000
id	0	1	2	3

Tabelle 5.5: Feature Set von Karte 1 in Gruppe 3

	Feature 1	Feature 2	Feature 3
Umfang	0.5151515	0.3030303	0.0000000
Länge	0.4587603	0.3456033	0.0000000
Breite	0.9034140	0.9097295	0.6443887
Fläche	0.8333333	0.2000000	0.0666667
<i>Hu1</i>	0.2666084	0.4120514	0.0000000
<i>Hu2</i>	0.1157352	0.1681231	0.0000000
Position x	37.0000000	24.0000000	20.0000000
Position y	34.0000000	26.0000000	10.0000000
id	0	3	3

Tabelle 5.6: Feature Set von Karte 2 in Gruppe 3

In Gruppe 1 wurden Hu-Momente nicht als Deskriptor verwendet, aber die Ergebnisse zeigen, dass die Bestimmung der Winkelabweichung korrekt ist. Es wurde jedoch durch Experiment festgestellt, dass bei einer Winkelabweichung von 45° , 135° , 225° oder 315° zwischen den beiden Labyrinthen das Bild die größte Verformung aufwies, was dazu führte, dass der Merkmalsabgleich fehlschlug, weshalb die Hu-Momente zur Gruppe 2 hinzugefügt wurden. *Hu1* und *Hu2* funktionierten zwar, aber der Fehler nahm zu, wenn Hu-Momente höherer Ordnung angewendet wurden.

In den Gruppen 2 und 3 ist zu erkennen, dass in Gruppe 2 die *id*-Zuweisungen in der Tabelle trotz gleicher Karten nicht genau übereinstimmen, aber wie aus 5.14 in der Abbildung ersichtlich ist, hat dies keinen signifikanten Einfluss auf die Berechnung der Winkelabweichung. Für die gleiche Winkelabweichung, aber für unterschiedliche Karten (siehe Gruppe 2 und Gruppe 3), unterscheiden sich die Standardabweichungen der Kurven in Gruppe 2 in der gleichen Größenordnung wie in Gruppe 3 (siehe Tabelle 5.7). Darüber hinaus ist auch aus Tabelle 5.7 ersichtlich, dass die MAPE von Gruppe 1 und Gruppe 2 sich von Gruppe 3 unterscheidet. Daher kann die Standardabweichung und MAPE zum Approximieren der Ähnlichkeit und Winkelabweichung verwendet werden.

	Gruppe 1	Gruppe 2	Gruppe 3
$MAPE(ed_1, ed_2)$	0.17 %	0.02 %	0.32 %
$MAPE(A_1, A_2)$	0.13 %	0.17 %	0.68 %
Standardabweichung	0.6825883932324971	0.7715407073995874	0.41237692495464046

Tabelle 5.7: Resultierender MAPE und Standardabweichung

5.5 Weitere Variante

Der Roboter kann auch mit anderen Mechanismen ausgestattet werden. Solange der eingestellte zugängliche Bereich entsprechend angepasst wird, benötigt der Roboter kein kinematisches Modell zur Unterstützung der Exploration der Umgebung. Natürlich sollte darauf geachtet werden, dass Singularitäten möglichst nicht auftreten werden, was anhand von Abbildung 5.18 eingestellt werden kann. Die folgende Abbildung zeigt ein Beispiel für einen Roboter, der an einen anderen Körper angepasst ist und die Karte, die er im *C Space* erstellt hat.

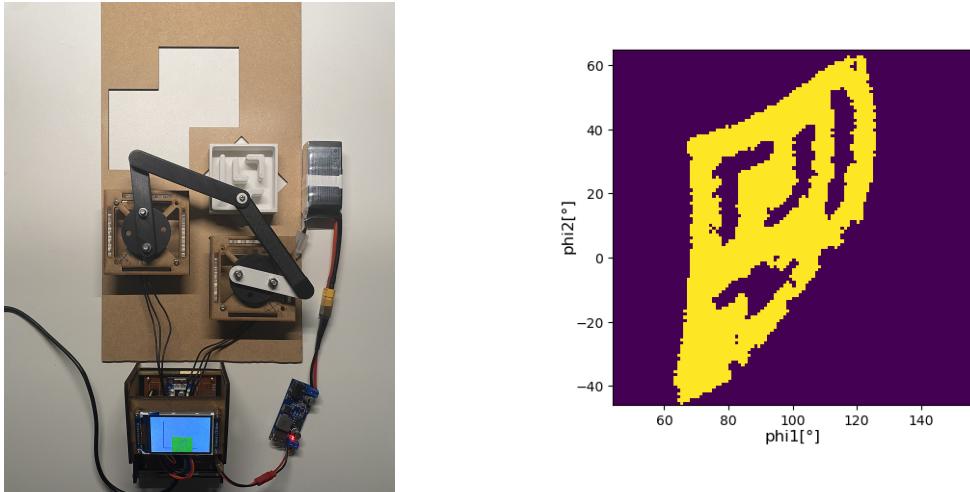


Abbildung 5.18: Eine der möglichen Varianten des Roboters bei Modus 2 und der entsprechende *C Space*

5.6 Zusammenfassung und Ausblick

In Modus 2 wird ein haptischer Explorationsalgorithmus implementiert, der auf die beiden Bedingungen des Fehlens der Sensoren und des kinematischen Modells des Robotermechanismus basiert. In diesem Algorithmus wird eine naive strombasierte Kollisionserkennung zur Markierung der Position von Hindernissen eingesetzt und MRC wird zur Steuerung der Roboteraktionen und des vom Servomotor zurückgegebenen Winkels zur Positionierung des Roboters im *C Space* verwendet. Dieser Ansatz demonstriert die Effektivität dieses Explorationsalgorithmus in einer unbekannten Umgebung von $60\text{mm} \times 60\text{mm}$ und beweist, dass die Methode in der Lage ist, die Umgebung mit verschiedenen Positionen und unbekannten kinematischen Modellen des Roboters nach einer kleinen Anpassung zu explorieren und zu modellieren.

Darüber hinaus wird festgestellt, dass einfache Deskriptoren zur Unterscheidung zwischen verschiedenen Hindernissen und damit zur weiteren Unterscheidung zwischen Karten verwendet werden können. Die Fähigkeit der Methode, sich an verschiedene Karten anzupassen, die durch unterschiedliche Mechanismen des Roboters erzeugt werden, ist jedoch noch nicht abschließend geklärt und bedarf weiterer Forschung und Validierung.

Insgesamt bietet der in dieser Forschung verwendete haptische Explorationsansatz, der auf keinen Sensoren und keinem Modell des Roboters basiert, eine alternative Lösung für die Exploration und Modellierung in unbekannten Umgebungen.

Da der Explorationsalgorithmus auf dem MRC basiert, das ein „reactive“ Ansatz ist, ist das Verhalten des Roboters nicht gut vorhersagbar, kann sie dazu führen, dass der Roboter eine Position wieder-

holt erkundet und damit Zeit vergeudet, was relativ ineffizient ist. (Die Erkundungszeit im Raum mm 60×60 mm beträgt 15 Minuten)

Wenn der „model-based“ Ansatz, der die Bahnplanungsmethode in Navigationsroboter [24] verwendet wird, in die Steuerung eines virtuellen Roboters im Modus 2 implementiert werden kann, kann dies dem Roboter ermöglichen, sein Verhalten effizient zu erkunden. Die Machbarkeit dieses Ansatzes kann ebenfalls untersucht werden.

Was den zugänglichen Bereich im *C Space* anbelangt, so würde der Roboter ein breiteres Anwendungsspektrum hat, wenn der Bereich auf einen unbegrenzten erweitert werden kann oder der Roboter bei unterschiedlichem Mechanismus ohne Anpassung die Umgebung explorieren kann. Die Vermeidung von Singularitäten unter solchen Bedingungen ist jedoch eine große Herausforderung. Ferner ist es auch eine offene Frage, wie dieser Ansatz auf Roboter mit höheren Freiheitsgraden ausweiten kann.

Beim Kartenvergleich kann die id-Zuweisung als ein zentrales Element angesehen werden. Während der Entwicklung hat die Wahl eines guten Deskriptors einen großen Einfluss auf die Genauigkeit der id-Zuweisung. Darüber hinaus kann die Komplexität (z. B. mehr Hindernisse) der Umgebung die Genauigkeit der id-Zuweisung ebenfalls beeinflussen. Dieses Problem kann durch die Wahl eines genaueren Merkmalsdeskriptors und dessen Implementierung in einem Controller mit höherem Aufwand, überwunden werden.

Literaturverzeichnis

- [1] Avr221: Discrete pid controller on tinyavr and megaavr devices. 2016.
- [2] Stefan Bethge. Erkennung von Kollisionen von Robotergliedmaßen anhand von Motor- und Sensordaten, 2011. Studienarbeit.
- [3] Valentino Braatenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA, 1984.
- [4] Daniel Butters, Emil T. Jonasson, Robert Stuart-Smith, and Vijay M. Pawar. Efficient environment guided approach for exploration of complex environments. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 929–934, 2019.
- [5] Lin Cheng, Jiayu Liu, Juyi Li, and Jianzhong Tang. Sensorless collision detection for a 6-dof robot manipulator.
- [6] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70, 04 2017.
- [7] Tuong Hoang, Trung Vuong, and Bang Pham. Study and development of parallel robots based on 5-bar linkage. 10 2015.
- [8] Zhihu Huang and J. Leng. Analysis of hu’s moment invariants on image scaling and rotation. volume 7, pages V7–476, 05 2010.
- [9] Austin Hughes. *Electric Motors and Drives*. Elsevier Ltd., USA, 3st edition, 2006.
- [10] Yoonkyu Hwang, Yuki Minami, and Masato Ishikawa. Virtual torque sensor for low-cost rc servo motors based on dynamic system identification utilizing parametric constraints. *Sensors*, 18(11), 2018.
- [11] Martin Hülse and Frank Pasemann. Dynamical neural schmitt trigger for robot control. pages 783–788, 08 2002.
- [12] M Fikret. Kanniah, Jagannathan. Ercan and Acosta Calderon, Carlos A. *Practical Robot Design: Game Playing Robots*. CRC Press, 1st edition, 2013.
- [13] Yasuhiro Kato, Pietro Balatti, Juan Gandarias, Mattia Leonori, Toshiaki Tsuji, and Arash Ajoudani. A self-tuning impedance-based interaction planner for robotic haptic exploration. *IEEE Robotics and Automation Letters*, 7:1–8, 10 2022.
- [14] Johannes Kilian. Simple image analysis by moments, 2001.
- [15] Guillermo Jesús Laguna Mosqueda. Exploration of an unknown environment with a differential drive disc robot. Master’s thesis, Computer Science Department, CIMAT, Guanajuato, Mexico, 10 2013.

- [16] David Charles Lee. The map-building and exploration strategies of a simple, sonar-equipped, mobile robot: An experimental, quantitative evaluation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 23–30, Minneapolis, MN, USA, April 1996. IEEE.
 - [17] Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, USA, 1st edition, 2017.
 - [18] Eitan Marder-Eppstein. rotate_recovery. Available online: http://wiki.ros.org/rotate_recovery, 2019. Zugriff am 6. März 2023.
 - [19] Frank Pasemann, Martin Hülse, and Keyan Ghazi-Zahedi. Evolved neurodynamics for robot control. pages 439–444, 01 2003.
 - [20] Gavin Paul, Stephen Webb, Dikai Liu, and Gaminini Dissanayake. Autonomous robot manipulator-based exploration and mapping system for bridge maintenance. *Robotics and Autonomous Systems*, 59:543–554, 07 2011.
 - [21] ROBOTIS. OpenCM 9.04 controller, n.d.
 - [22] Leonardo Romero, Eduardo F. Morales, and L. Enrique Sucar. Exploration and navigation for mobile robots with perceptual limitations. *International Journal of Advanced Robotic Systems*, 3(3):36, 2006.
 - [23] Tim Schneider, Boris Belousov, Georgia Chalvatzaki, Diego Romeres, Devesh K. Jha, and Jan Peters. Active exploration for robotic manipulation, 2022.
 - [24] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. The MIT Press, 2nd edition, 2011.
 - [25] Lila Torabi, Moslem Kazemi, and Kamal Gupta. Configuration space based efficient view planning and exploration with occupancy grids. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2827–2832, 2007.
 - [26] Yiming Wang, Stuart James, Ellie Stathopoulou, Carlos Beltran-Gonzalez, Yoshinori Konishi, and Alessio Del Bue. Autonomous 3-d reconstruction, mapping, and exploration of indoor environments with a robotic stange. *IEEE Robotics and Automation Letters*, PP:1–1, 07 2019.
 - [27] Michael A. Wirth. Shape analysis and measurement. 2004.
 - [28] B. Yamauchi, A. Schultz, and W. Adams. Mobile robot exploration and map-building with continuous localization. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, volume 4, pages 3715–3720 vol.4, 1998.
 - [29] Bowei Zhang and Pengcheng Liu. Model-based and model-free robot control: A review. In *Proceedings of the 8th International Conference on Robot Intelligence Technology and Applications*, Cardiff, UK, December 2020. IEEE.
 - [30] Kaiyu Zheng. ROS navigation tuning guide. *CoRR*, abs/1706.09068, 2017.
-

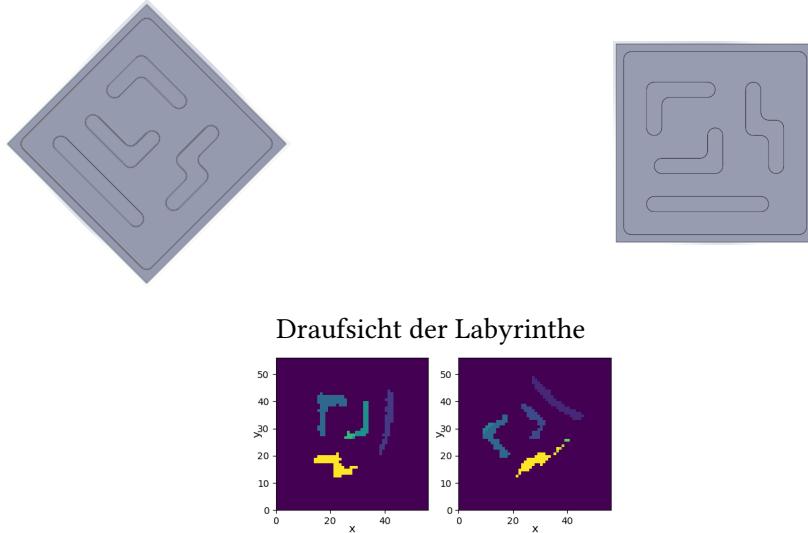
Anhang A

Anhang

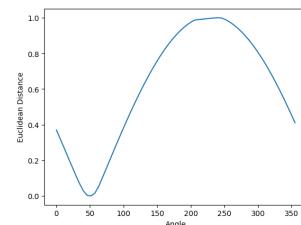
A.1 Zusätzliche Ergebnisse von Modus2

A.1.1 Beispiele für andere Kartenvergleich

Beispiel 1



Die verarbeitete und markierte Karten mit 45-Grad-Winkelabweichung

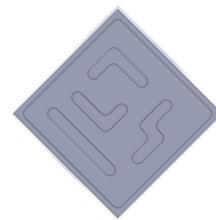
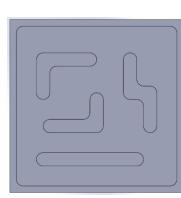


Die normalisierte minimale Summe des euklidischen Abstands über Rotation von 0° bis 360°
• θ_{min} liegt bei 51°

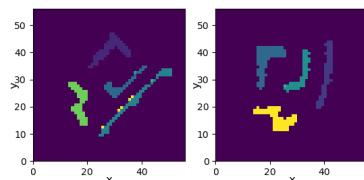
$MAPE(ed_1, ed_2)$	0.04 %
$MAPE(A_1, A_2)$	0.05 %

Resultierender MAPE

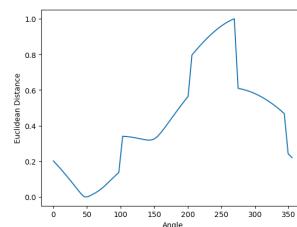
Beispiel 2



Draufsicht der Labyrinthe



Die verarbeitete und markierte Karten mit 45-Grad-Winkelabweichung



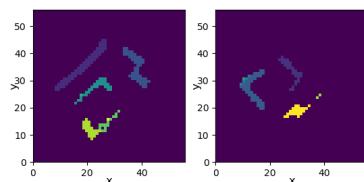
Die normalisierte minimale Summe des euklidischen Abstands über Rotation von 0° bis 360°
 . θ_{min} liegt bei 46°

$MAPE(ed_1, ed_2)$	0.01 %
$MAPE(A_1, A_2)$	0.37 %

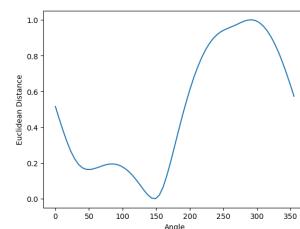
Resultierender MAPE

Beispiel 3

Draufsicht der Labyrinthe



Die verarbeitete und markierte Karten mit -90-Grad-Winkelabweichung

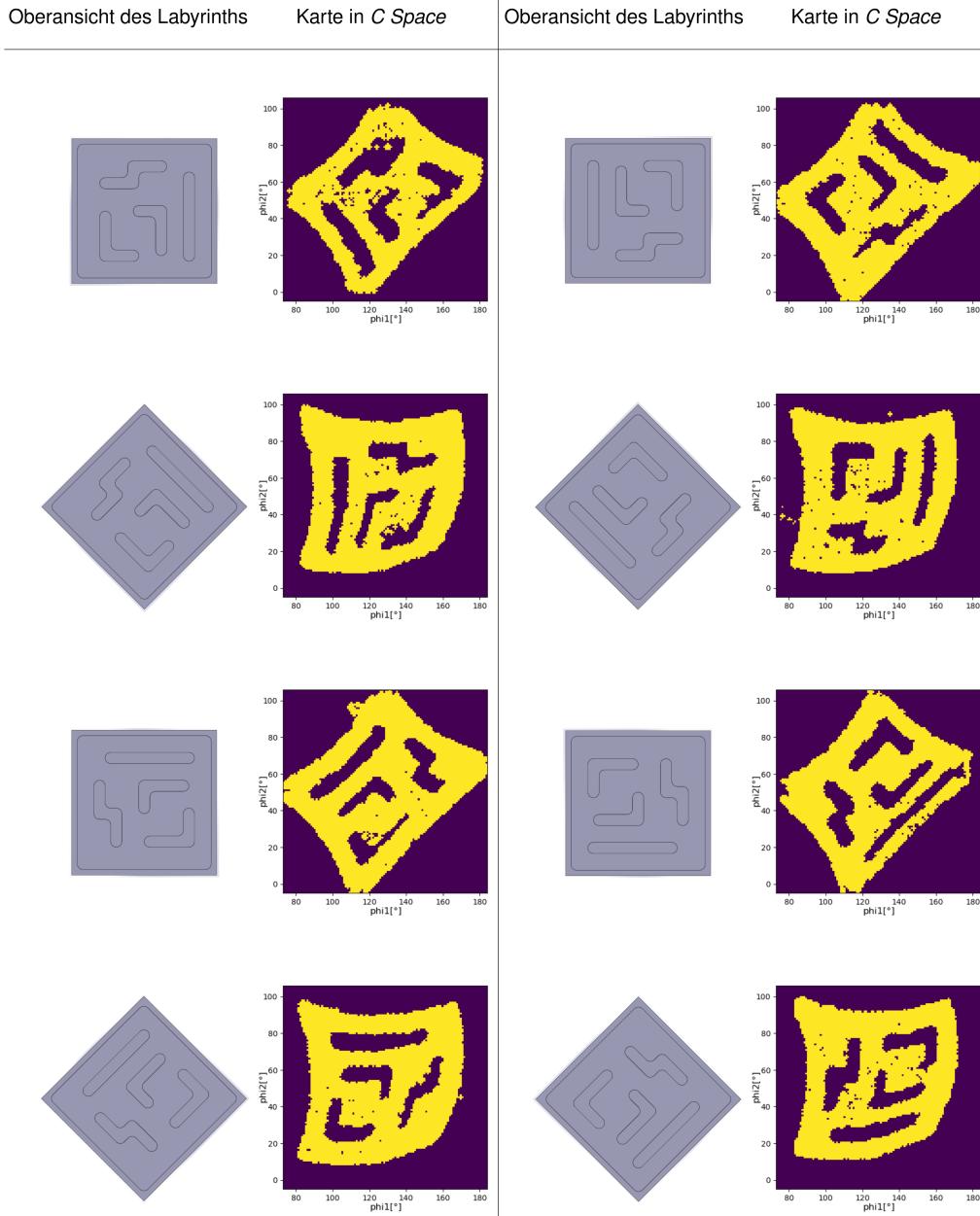


Die normalisierte minimale Summe des euklidischen Abstands über Rotation von 0° bis 360°
 $\cdot \theta_{min}$ liegt bei 149°

$MAPE(ed_1, ed_2)$	0.32 %
$MAPE(A_1, A_2)$	0.95 %

Resultierender MAPE

A.1.2 Alle in 45-Grad-schrittweise erstellte Karten



B.2 Quellcode für Modus 1

Definition of Constants

```
#define M_PI 3.14159265358979323846          // Constant pi.
#define DXL_SERIAL Serial1                     // Serial port to communicate with the Dynamixel motor controller.
#define DEBUG_SERIAL Serial                      // Serial port to output debug messages.
#define DXL_DIR_PIN 28                         // Direction pin for the motor controller.
#define LED_PIN 14                            // Pin for the LED on the microcontroller board.
#define period1 1000000                        // Period of the first task in microseconds.
#define period2 40000                          // Period of the second task in microseconds.
#define Input_Voltage_add 144                  // Address of the input voltage in the Dynamixel protocol.
#define Input_Voltage_size 2                   // Size of the input voltage in bytes in the Dynamixel protocol.
#define Timeout 10                           // Timeout for communication with the Dynamixel motor controller.
#define Motor_Num 2                           // Number of motors being controlled.
#define Max_ang_v (50.0F)                     // Maximum angular velocity constraint for the motors.
#define Min_ang_v (-50.0F)                    // Minimum angular velocity constraint for the motors.

#define Kp1 (0.7F)                           // Proportional gain for motor 1
#define Ki1 (0.0F)                           // Integral gain for motor 1
#define Kd1 (0.25F)                          // Derivative gain for motor 1

#define Kp2 (0.7F)                           // Proportional gain for motor 2
#define Ki2 (0.0F)                           // Integral gain for motor 2
#define Kd2 (0.25F)                          // Derivative gain for motor 2
```

Loop 1

```
// Check if it's time to run the first task
if (currentMicros - previousMicros1 >= period1) {
    uint16_t input_voltage = 0;
    // Read the input voltage of Motor 1 and store it in voltage1
    dxl.read(Motor_1_ID, Input_Voltage_add, Input_Voltage_size, (uint8_t*)&input_voltage, sizeof(input_voltage), Timeout);
    voltage1 = input_voltage;
    // Read the input voltage of Motor 2 and store it in voltage2
    input_voltage = 0;
    dxl.read(Motor_2_ID, Input_Voltage_add, Input_Voltage_size, (uint8_t*)&input_voltage, sizeof(input_voltage), Timeout);
    voltage2 = input_voltage;
    // Update the previousMicros1 variable to keep track of when to run this task again
    previousMicros1 += period1;
}
```

Loop 2

```

// Check if it's time to run the second task
if (currentMicros - previousMicros2 >= period2) {
    // Update the previousMicros2 variable for timing purposes
    previousMicros2 += period2;

    // Call the visual_robot function to clear the previous visualization of robot's heading
    visual_robot(heading, 1);

    /* Get essential data */
    // Get present left present velocity and right present velocity
    float ang1_v = dxl.getPresentVelocity(Motor_1_ID, UNIT_RPM);
    float ang2_v = dxl.getPresentVelocity(Motor_2_ID, UNIT_RPM);

    // Get present left present position and right present position
    float present_position_1 = dxl.getPresentPosition(Motor_1_ID, UNIT_DEGREE);
    float present_position_2 = dxl.getPresentPosition(Motor_2_ID, UNIT_DEGREE);

    // Get present left present current and right present current
    float current_1 = abs(dxl.getPresentCurrent(Motor_1_ID, UNIT_MILLI_AMPERE));
    float current_2 = abs(dxl.getPresentCurrent(Motor_2_ID, UNIT_MILLI_AMPERE));

    // Apply an IIR filter to the present current readings
    f_current_1 = (1 - alpha1) * current_1 + alpha1 * f_current_1;
    f_current_2 = (1 - alpha1) * current_2 + alpha1 * f_current_2;

    // Apply an IIR filter to the present velocity readings
    f_ang1_v = (1 - alpha2) * ang1_v + alpha2 * f_ang1_v;
    f_ang2_v = (1 - alpha2) * ang2_v + alpha2 * f_ang2_v;

    // Convert the present position readings from degrees to integers
    int16_t phi1 = round(fmod(present_position_1 + 135, 360.0));
    int16_t phi2 = round(fmod(present_position_2 + 45, 360.0));
    phi1 = (phi1 < 0) ? phi1 + 360 : phi1;
    phi2 = (phi2 < 0) ? phi2 + 360 : phi2;

    /* Motion Planning */
    phi1 = (phi2 > 270 && phi1 < 90) ? phi1 + 360 : phi1; // adjust phi1 if needed to make the motion smoother
    phi2 = (phi1 > 270 && phi2 < 90) ? phi2 + 360 : phi2; // adjust phi2 if needed to make the motion smoother
    double error_1 = (phi2 - phi1) * 0.5; // calculate error for phi1
    double error_2 = (phi1 - phi2) * 0.5; // calculate error for phi2

    // PID Controller for phi1
    double p_control_1 = Kp1 * error_1; // calculate proportional control for phi1
    double d_control_1 = Kd1 * (error_1 - previous_error_1); // calculate derivative control for phi1
    // i_control_1 = i_control_1 + Ki1 * error_1; // calculate integral control for phi1, currently not used
    double d_phi1 = p_control_1 + d_control_1 + i_control_1; // calculate final control signal for phi1
    previous_error_1 = error_1; // store previous error for phi1

    // PID Controller for phi2
    double p_control_2 = Kp2 * error_2; // calculate proportional control for phi2
    double d_control_2 = Kd2 * (error_2 - previous_error_2); // calculate derivative control for phi2
    // i_control_2 = i_control_2 + Ki2 * error_2; // calculate integral control for phi2, currently not used
    double d_phi2 = p_control_2 + d_control_2 + i_control_2; // calculate final control signal for phi2
    previous_error_2 = error_2; // store previous error for phi2
    /* Motion Planning */

    /* Motor Control */
    d_phi1 = (d_phi1 > Min_ang_v) ? ((d_phi1 < Max_ang_v) ? d_phi1 : Max_ang_v) : Min_ang_v; // Clamp motor 1's velocity
    d_phi2 = (d_phi2 > Min_ang_v) ? ((d_phi2 < Max_ang_v) ? d_phi2 : Max_ang_v) : Min_ang_v; // Clamp motor 2's velocity
    dxl.setGoalPWM(Motor_1_ID, d_phi1 * (50.0 / voltage1), UNIT_PERCENT); // Set the goal PWM for motor 1
    dxl.setGoalPWM(Motor_2_ID, d_phi2 * (50.0 / voltage2), UNIT_PERCENT); // Set the goal PWM for motor 2
    /* Motor Control */

    /* Visualization and Collision Detection */
    uint16_t phi1_w = fmod(phi1, 360) / 2; // convert phi1 to a value between 0 and 180 for visualization
    uint16_t phi2_w = fmod(phi2, 360) / 2; // convert phi2 to a value between 0 and 180 for visualization

    // Current based collision detection
    if ((f_current_1 > 25 && f_current_2 > 25) && abs(f_current_1 - f_current_2) < 5) {
        // If there is a collision, set the collision flag to 1 and turn on the LED
        collision = 1;

```

```
digitalWriteFast(LED_PIN, LOW); // Turn on the LED
// Visualize obstacle on screen
Paint_DrawPoint(collision_x, collision_y, WHITE, DOT_PIXEL_2X2, DOT_FILL_AROUND);
double angle = M_PI * ((450 - phi1) / 180.0);
collision_x = 120 + (visual_bar + 5) * cos(angle);
collision_y = 280 + (visual_bar + 5) * sin(angle);
Paint_DrawPoint(collision_x, collision_y, BLACK, DOT_PIXEL_2X2, DOT_FILL_AROUND);
}
else {
    // If there is no collision, set the collision flag to 0 and turn off the LED
    collision = 0;
    digitalWriteFast(LED_PIN, HIGH); // Turn off the LED
}
/* Visualization and Collision Detection */
}
```

C.3 Quellcode für Modus 2

Definition of Constants

```

#define M_PI 3.14159265358979323846          // Constant pi.
#define DXL_SERIAL Serial1                     // Serial port to communicate with the Dynamixel motor controller.
#define DEBUG_SERIAL Serial                    // Serial port to output debug messages.
#define DXL_DIR_PIN 28                         // Direction pin for the motor controller.
#define LED_PIN 14                            // Pin for the LED on the microcontroller board.
#define period1 10000000                      // Period of the first task in microseconds
#define period2 40000                          // Period of the second task in microseconds
#define Input_Voltage_add 144                  // Address of the input voltage in the Dynamixel protocol.
#define Input_Voltage_size 2                   // Size of the input voltage in bytes in the Dynamixel protocol.
#define Timeout 10                           // Timeout for communication with the Dynamixel motor controller.
#define Motor_Num 2                           // Number of motors being controlled.
#define Max_ang_v (10.0F)                     // Maximum angular velocity constraint for the motors.
#define Min_ang_v (-10.0F)                    // Minimum angular velocity constraint for the motors.
#define Explore_v 8                           // Maximum translational velocity for the virtual robot.
#define max_dis 21                           // Maximum sensible distance for the virtual robot.
#define dt (0.1F)                            // Time step for virtual mobile robot
#define Motor_1_ID 3                          // ID of the first motor being controlled.
#define Motor_2_ID 4                          // ID of the second motor being controlled.
#define alpha1 (0.8F)                         // Constant value for the IIR filter for current.
#define alpha2 (0.6F)                         // Constant value for the IIR filter for velocity.
#define Range 56                             // Half size of work area(the area to be explored; O_map and F_map).
%
#define cs_length 112                        // Full size of work area(the area to be explored; O_map and F_map).
%
#define Center1 129                         // Center of work area in horizontal-axis(phi1).
#define Center2 51                           // Center of work area in Vertical-axis(phi2).
#define shift_x (Center1 - Range)           // Horizontal shift from C Space work area(O_map and F_map).
#define shift_y (Center2 - Range)           // Vertical shift from C Space work area(O_map and F_map).
#define Cell_bits 1                          // Number of bits per cell in the map(O_map and F_map).
#define Reg_size 8                           // Size of the register to save the map(O_map and F_map).
#define Reg_partition 8                     // Number of cells per register (Reg_size / Cell_bits).
#define Max_shift 7                           // Maximum shift value for a register (Reg_partition - 1).
#define DWNS_img_size 56                     // Size of the downsampling image for feature extraction.
%
#define Valid_descriptor 6                  // Number of valid descriptors for feature matching.
%
#define Max_features 7                      // Maximum number of features for feature matching.
#define Max_maps 2                          // Maximum number of maps for feature matching.
#define Map_size (int)((2 * Range) * (2 * Range)) / Reg_partition // Size of work area(O_map and F_map).

uint8_t O_Map[Map_size] = {0};                //O_Map
uint8_t F_Map[Map_size] = {0};                //F_Map
feature feature_set[Max_maps][Max_features]; //features set (Max_map x Max_feature)
double Data_Points[Max_maps][3][Max_features]; //small features set will only save position and id of features
uint8_t DWNS_img[DWNS_img_size * DWNS_img_size];//Downsampling image, bit 0 for pixel value, bit 1-7 for storing label

typedef struct {
    double descriptor[Valid_descriptor]; // A double array with size Valid_descriptor.
    uint8_t position_x;                 // x position of the feature's centroid.
    uint8_t position_y;                 // y position of the feature's centroid.
    int8_t id;                         // feature's ID. -1 for non-exist
} feature;

```

C.3.1 Exploration

Read und Write of O_{Map} and F_{Map}

```

uint8_t read_map(int arg1, int arg2, uint8_t* map_ptr) {
    // Clamp the coordinates to within the boundaries of the map.
    arg1 = (arg1 >= 0) ? (arg1 < cs_length) ? arg1 : cs_length - 1 : 0;
    arg2 = (arg2 >= 0) ? (arg2 < cs_length) ? arg2 : cs_length - 1 : 0;

    // Compute the index into the map's array.
    int temp = (int)(arg1 / (Reg_size / Cell_bits)) + ((cs_length) / (Reg_size)) * arg2;

    // Compute the shift required to extract the value of the pixel.
    uint8_t shift = (Reg_partition - 1 - arg1 % Reg_partition);

    // Extract and return the value of the pixel.
    return (map_ptr[temp] & (1 << shift)) >> shift;
}

void write_map(int arg1, int arg2, uint8_t value, uint8_t* map_ptr) {
    // Clamp the coordinates to within the boundaries of the map.
    arg1 = (arg1 >= 0) ? (arg1 < cs_length) ? arg1 : cs_length - 1 : 0;
    arg2 = (arg2 >= 0) ? (arg2 < cs_length) ? arg2 : cs_length - 1 : 0;

    // Compute the index into the map's array.
    int temp = (int)(arg1 / (Reg_size / Cell_bits)) + ((cs_length) / (Reg_size)) * arg2;

    // Compute the shift required to set the value of the pixel.
    uint8_t shift = (Reg_partition - 1 - arg1 % Reg_partition);

    // Set or clear the value of the pixel depending on the input value.
    if (value == 1) {
        map_ptr[temp] |= (1 << shift);
    } else {
        map_ptr[temp] &= ~(1 << shift);
    }
}

```

Distance Sensor

```

void distance_sensor(int16_t pos_x, int16_t pos_y, float heading, double* l_value, double* r_value){
    // four angles in which to measure distance to obstacles
    float angle[4] ={M_PI *(15 / 180.0), M_PI *(10 / 180.0), M_PI *(-10 / 180.0), M_PI *(-15 / 180.0)};

    // Initialize the distance values for each angle to the maximum distance
    double distance[4] ={max_dis, max_dis, max_dis, max_dis};

    // Declare variables to store the position of each sensor
    int16_t x_sens, y_sens;

    // Iterate through each angle
    for(uint8_t i = 0; i < 4; i++){
        // position of the sensor at the current angle
        x_sens = pos_x + max_dis * cos(heading + angle[i]);
        y_sens = pos_y + max_dis * sin(heading + angle[i]);

        // Iterate through each point along the line between the robot and the sensor
        for(uint8_t j = 0; j < max_dis; j++){
            // position of the current point
            float u = j / float(max_dis);
            int16_t x = round(pos_x *(1 - u) + x_sens * u);
            int16_t y = round(pos_y *(1 - u) + y_sens * u);

            // Check if the current point is an obstacle
            if(read_map(x, y, O_Map) == 1){
                // distance to the obstacle and store it
                distance[i] = sqrt((pos_x - x) *(pos_x - x) +(pos_y - y) *(pos_y - y));
                break;
            }
        }
    }

    // average distance to obstacles on the left and right sides of the robot
    double temp_l =(distance[0] + distance[1]) / 2.0;
    double temp_r =(distance[2] + distance[3]) / 2.0;

    // Add random noise to the distance values, but only if they are greater than 0.2
    (*l_value) = (temp_l > 0.2) ? (temp_l + random(-200, 200) / 100.0) : 0.0;
    (*r_value) = (temp_r > 0.2) ? (temp_r + random(-200, 200) / 100.0) : 0.0;
}

```

Loop 1

```

// Check if it's time to run the first task
if (currentMicros - previousMicros1 >= period1) {
    uint16_t input_voltage = 0;
    // Read the input voltage of Motor 1 and store it in voltage1
    dxl.read(Motor_1_ID, Input_Voltage_add, Input_Voltage_size, (uint8_t*)&input_voltage, sizeof(input_voltage), Timeout);
    voltage1 = input_voltage;
    // Read the input voltage of Motor 2 and store it in voltage2
    input_voltage = 0;
    dxl.read(Motor_2_ID, Input_Voltage_add, Input_Voltage_size, (uint8_t*)&input_voltage, sizeof(input_voltage), Timeout);
    voltage2 = input_voltage;

    // Loop through the obstacle map and paint any obstacles detected
    for (uint8_t i = 0; i < Range * 2; i++) {
        for (uint8_t j = 0; j < Range * 2; j++) {
            if (read_map(j, i, O_Map)) {
                Paint_SetPixel(30 + i + (Center2 - Range), 70 + j + (Center1 - Range), RED);
            }
        }
    }
    // Create borders at the edge of the obstacle map to avoid robot driving out of the work area
    for (uint8_t i = 0; i < Range * 2; i++) {
        write_map(0, i, 1, O_Map);
        write_map(Range * 2 - 1, i, 1, O_Map);
        write_map(i, Range * 2 - 1, 1, O_Map);
        write_map(i, 0, 1, O_Map);
    }
    // Update the previousMicros1 variable to keep track of when to run this task again
    previousMicros1 += period1;
}

```

Loop 2

```

// Check if it's time to run the second task
if (currentMicros - previousMicros2 >= period2) {
    // Update the previousMicros2 variable for timing purposes
    previousMicros2 += period2;

    // Call the visual_robot function to clear the previous visualization of robot's heading
    visual_robot(heading, 1);

    /* Get essential data */
    // Get present left present velocity and right present velocity
    float ang1_v = dxl.getPresentVelocity(Motor_1_ID, UNIT_RPM);
    float ang2_v = dxl.getPresentVelocity(Motor_2_ID, UNIT_RPM);

    // Get present left present position and right present position
    float present_position_1 = dxl.getPresentPosition(Motor_1_ID, UNIT_DEGREE);
    float present_position_2 = dxl.getPresentPosition(Motor_2_ID, UNIT_DEGREE);

    // Get present left present current and right present current
    float current_1 = abs(dxl.getPresentCurrent(Motor_1_ID, UNIT_MILLI_AMPERE));
    float current_2 = abs(dxl.getPresentCurrent(Motor_2_ID, UNIT_MILLI_AMPERE));

    // Apply an IIR filter to the present current readings
    f_current_1 = (1 - alpha1) * current_1 + alpha1 * f_current_1;
    f_current_2 = (1 - alpha1) * current_2 + alpha1 * f_current_2;

    // Apply an IIR filter to the present velocity readings
    f_ang1_v = (1 - alpha2) * ang1_v + alpha2 * f_ang1_v;
    f_ang2_v = (1 - alpha2) * ang2_v + alpha2 * f_ang2_v;

    // Convert the present position readings from degrees to integers
    int16_t phi1 = round(fmod(present_position_1 + 135, 360.0));
    int16_t phi2 = round(fmod(present_position_2 + 45, 360.0));
    phi1 = (phi1 < 0) ? phi1 + 360 : phi1;
    phi2 = (phi2 < 0) ? phi2 + 360 : phi2;

    // Apply shift from C Space coordinates to the work area coordinates
    int8_t pos_x = (phi1 >= 360 + shift_x) ? phi1 - 360 - shift_x : phi1 - shift_x;
    int8_t pos_y = (phi2 >= 360 + shift_y) ? phi2 - 360 - shift_y : phi2 - shift_y;
    pos_x = (pos_x >= 0) ? (pos_x < cs_length) ? pos_x : cs_length - 1 : 0;
    pos_y = (pos_y >= 0) ? (pos_y < cs_length) ? pos_y : cs_length - 1 : 0;

    // Perform collision detection and set the collision variable accordingly
    uint8_t collision = 0;
    collision = (f_current_1 > 17) << 0 | (f_current_2 > 17) << 4;
    /* Get essential data */

    /* Labelization */
    // Labelize obstacle in obstacle map if a collision has been detected
    if (collision != 0) {
        // Check for obstacles within 5 units of the robot and label them in the obstacle map
        x_sens = pos_x + 5 * cos(heading);
        y_sens = pos_y + 5 * sin(heading);
        for (int8_t i = -1; i < 2; i++) {
            uint8_t x = int(x_sens + i * cos(heading + 1.57));
            uint8_t y = int(y_sens + i * sin(heading + 1.57));
            if (read_map(x, y, F_Map) != 1) {
                write_map(x, y, 1, O_Map);
            }
        }
    }

    // Clear the robot's nearby area in the obstacle map to make it easier to navigate
    write_map(pos_x, pos_y + 1, 0, O_Map);
    write_map(pos_x, pos_y - 1, 0, O_Map);
    write_map(pos_x, pos_y, 0, O_Map);
    write_map(pos_x - 1, pos_y, 0, O_Map);
    write_map(pos_x + 1, pos_y, 0, O_Map);

    // Labelize the robot's nearby area as free in the free map
    write_map(pos_x, pos_y + 1, 1, F_Map);

```

```

write_map(pos_x, pos_y - 1, 1, F_Map);
write_map(pos_x, pos_y, 1, F_Map);
write_map(pos_x - 1, pos_y, 1, F_Map);
write_map(pos_x + 1, pos_y, 1, F_Map);
/* Labelization */

/* Motion Planning */
// Perception
// Use distance sensors to detect obstacles and determine the robot's surroundings
double input_l = 0;
double input_r = 0;
distance_sensor(pos_x, pos_y, heading, &input_l, &input_r);

// Minimal Recurrent Controller
input_l = tanh(0.4 * (12.0 - input_l)); // -1 free, 1 occupied; dis_max free, 0 occupied
input_r = tanh(0.4 * (12.0 - input_r)); // -1 free, 1 occupied; dis_max free, 0 occupied
double act_l = -7.0 * input_r + 3.0 * neu_l - 4.0 * neu_r;
double act_r = -7.0 * input_l + 3.0 * neu_r - 4.0 * neu_l;
neu_l = tanh(act_l);
neu_r = tanh(act_r);
/* Motion Planning */

/* Differential Drive Robot Kinematics */
float v = (Explore_v * (neu_r + neu_l)) / 2.0; // average velocity
v = (v > 1) ? v : 1; // Limit the minimum velocity to 1
float omega = (Explore_v * (neu_r - neu_l)) / 2.0; // angular velocity
heading += omega * dt; // Update the heading of the robot
heading = fmod(heading + 2.0 * M_PI, 2.0 * M_PI); // Ensure the heading is between 0 and 2PI
heading = (heading < 0) ? heading + 2.0 * M_PI : heading; // Ensure the heading is positive
d_phi1 = v * cos(heading); // velocity of motor 1
d_phi2 = v * sin(heading); // velocity of motor 2
/* Differential Drive Robot Kinematics */

/* Recovery behaviour */
// If the robot is stuck and one of the motor velocities is close to zero,
// add a small amount of rotation to the heading to try and get it unstuck
if ((abs(d_phi1) >= 0.1 && abs(f_ang1_v) <= 0.1) || (abs(d_phi2) >= 0.1 && abs(f_ang2_v) <= 0.1)) {
    heading += 0.05;
}
/* Recovery behaviour */

/* Motor Control */
d_phi1 = (d_phi1 > Min_ang_v) ? ((d_phi1 < Max_ang_v) ? d_phi1 : Max_ang_v) : Min_ang_v; // Clamp motor 1's velocity
d_phi2 = (d_phi2 > Min_ang_v) ? ((d_phi2 < Max_ang_v) ? d_phi2 : Max_ang_v) : Min_ang_v; // Clamp motor 2's velocity
dxl.setGoalPWM(Motor_1_ID, d_phi1 * (50.0 / voltage1), UNIT_PERCENT); // Set goal PWM for motor 1
dxl.setGoalPWM(Motor_2_ID, d_phi2 * (50.0 / voltage2), UNIT_PERCENT); // Set goal PWM for motor 2
/* Motor Control */

/* Visualization */
phi1 = (phi1 > 180) ? phi1 - 360 : phi1; // Ensure that phi1 is between -180 and 180
phi2 = (phi2 > 180) ? phi2 - 360 : phi2; // Ensure that phi2 is between -180 and 180
Paint_SetPixel(30 + phi2, 70 + phi1, GRAY); // Set a pixel in the visualization
visual_robot(heading, 0); // Update the visualization of the robot's current heading
/* Visualization */
}

```

C.3.2 Kartenvergleich

Perimeter

```
// Computes the perimeter of a shape in binary image.
uint16_t perimeter(uint8_t* input, uint8_t width, uint8_t height) {
    // Initialize the perimeter variable to zero.
    uint16_t perimeter = 0;

    // Loop through all pixels of the input image, excluding the outermost row and column.
    for (uint8_t i = 1; i < width - 1; i++) {
        for (uint8_t j = 1; j < height - 1; j++) {
            uint8_t kernel_sum = 0;

            // If the current pixel is a foreground pixel (value 1), compute the sum of its 4-neighbors.
            if (read_map(i, j, input) == 1) {
                kernel_sum += read_map(i + 0, j - 1, input);
                kernel_sum += read_map(i - 1, j + 1, input);
                kernel_sum += read_map(i + 0, j + 1, input);
                kernel_sum += read_map(i + 1, j + 0, input);

                // If the sum of the 4-neighbors is less than 4, increment the perimeter variable.
                if (kernel_sum < 4) {
                    perimeter++;
                }
            }
        }
    }

    // Iterate over the top, bottom, left, and right edges of the image.
    // If a foreground pixel is found, increment the perimeter variable.
    for (uint8_t j = 0; j < width; j++) {
        perimeter = (read_map(j, 0, input) == 1) ? (perimeter + 1) : perimeter;
    }
    for (uint8_t i = 0; i < height; i++) {
        perimeter = (read_map(0, i, input) == 1) ? (perimeter + 1) : perimeter;
    }
    for (uint8_t j = 0; j < width; j++) {
        perimeter = (read_map(j, height - 1, input) == 1) ? (perimeter + 1) : perimeter;
    }
    for (uint8_t i = 0; i < height; i++) {
        perimeter = (read_map(width - 1, i, input) == 1) ? (perimeter + 1) : perimeter;
    }

    // Return the final perimeter value.
    return perimeter;
}
```

Width and Length

```

// computes the width and height of the largest rectangular shape present in the binary image
void compute_shape_size(uint8_t* input, double* width, double* height) {
    // Initialize moment variables to zero
    double m00 = 0, m10 = 0, m01 = 0;
    uint8_t x, y, idx;

    // Compute image moments
    // Loop through all pixels of the input image
    for (y = 0; y < DWNS_img_size; y++) {
        for (x = 0; x < DWNS_img_size; x++) {
            idx = y * DWNS_img_size + x;
            // Check if pixel is white and update moment variables
            if (read_map(x, y, input)) {
                m00 += 1;
                m10 += x;
                m01 += y;
            }
        }
    }

    // Compute the centroid of the shape
    double x_mean = m10 / m00;
    double y_mean = m01 / m00;

    // Initialize central moment variables to zero
    double mu11 = 0, mu20 = 0, mu02 = 0;

    // Compute central moments
    // Loop through all pixels of the input image
    for (y = 0; y < DWNS_img_size; y++) {
        for (x = 0; x < DWNS_img_size; x++) {
            idx = y * DWNS_img_size + x;
            // Check if pixel is white and update central moment variables
            if (read_map(x, y, input)) {
                double x_diff = x - x_mean;
                double y_diff = y - y_mean;
                mu11 += x_diff * y_diff;
                mu20 += x_diff * x_diff;
                mu02 += y_diff * y_diff;
            }
        }
    }

    // Compute normalized central moments
    double u11 = mu11 / m00;
    double u20 = mu20 / m00;
    double u02 = mu02 / m00;

    // Compute semi-axes of the ellipse that fits the shape
    double a = sqrt((u20 + u02 + sqrt((u20 - u02) * (u20 - u02) + 4 * u11 * u11)) / 2);
    double b = sqrt((u20 + u02 - sqrt((u20 - u02) * (u20 - u02) + 4 * u11 * u11)) / 2);

    // Update the width and height pointers with the dimensions of the ellipse
    *width = a * 2;
    *height = b * 2;
}

```

Hu-Moments

```

// computes the (p + q) th moment of the shape in binary image.
double calcMoment(double p, double q, uint8_t* input, uint8_t width, uint8_t height) {
    // Initialize a variable to accumulate the sum of the moments.
    double sum = 0;
    // Loop through all pixels of the input image.
    for (uint8_t x = 0; x < width; x++) {
        for (uint8_t y = 0; y < height; y++) {
            // Extract the value of the pixel and use it to update the sum.
            sum += read_map(x, height - 1 - y, input) * pow(x, p) * pow(y, q);
        }
    }
    // Return the final sum of moments.
    return sum;
}

// calculates the central moment of the shape in binary image
double calcCentralMoment(double p, double q, uint8_t* input, uint8_t width, uint8_t height) {
    // Calculate the zeroth, first, and second order moments of the binary image
    double M00 = calcMoment(0, 0, input, width, height);
    double M10 = calcMoment(1, 0, input, width, height);
    double M01 = calcMoment(0, 1, input, width, height);
    // Calculate the x and y coordinates of the origin of the image
    double originOfX = M10 / M00;
    double originOfY = M01 / M00;
    double sum = 0;
    // Loop through all pixels of the input image and calculate the (p+q) th central moment
    for (uint8_t x = 0; x < width; x++) {
        for (uint8_t y = 0; y < height; y++) {
            sum += read_map(x, height - 1 - y, input) * pow((x - originOfX), p) * pow((y - originOfY), q);
        }
    }
    // Return the calculated moment
    return sum;
}

// calculates the normalized central moment of the shape in binary image
double calcNormalizedCentralMoment(double p, double q, uint8_t* input, uint8_t width, uint8_t height) {
    // Calculate the order of the moment
    double order = ((p + q) / 2.0) + 1;
    // Calculate the normalized moment using the origin moments
    double a = calcCentralMoment(p, q, input, width, height) / pow(calcCentralMoment(0, 0, input, width, height), order);
    // Return the calculated normalized moment
    return a;
}

// calculates the first two Hu moments of the shape in binary image
void getHuMoments(uint8_t* input, uint8_t width, uint8_t height, double* hu1, double* hu2) {
    // Calculate the second order normalized central moments
    double nu20 = calcNormalizedCentralMoment(2, 0, input, width, height);
    double nu02 = calcNormalizedCentralMoment(0, 2, input, width, height);
    double nu11 = calcNormalizedCentralMoment(1, 1, input, width, height);

    // Calculate the first two Hu moments using the normalized central moments
    *hu1 = nu20 + nu02;
    *hu2 = pow((nu20 - nu02), 2) + 4 * (pow(nu11, 2));
}

```

Connected Component Labeling

```

// extracts connected pixel
uint8_t connected_component_labeling(uint8_t* input, uint8_t* equ_list, uint8_t width, uint8_t height) {
    // bit 1-7 of *input will be used to store the label
    // bit 0 is the pixel value
    // O_Map will be used as a buffer of equivalent list

    uint8_t next_label = 1; // Initialize the next available label to 1

    // First pass: label connected components
    // Loop through all pixels of the downsampling image
    for (uint8_t y = 0; y < height; y++) {
        for (uint8_t x = 0; x < width; x++) {
            uint16_t index = height * y + x;

            // Check if the current pixel is part of a component
            if ((input[index] & 0b00000001) == 1) {
                uint8_t n_label = 0; // Initialize the neighboring label to 0

                // Check neighboring pixels for existing labels
                // Check left pixel
                if (x > 0) {
                    index = height * y + x - 1;
                    uint8_t data = input[index];
                    if (data & 0b00000001 == 1) {
                        data = data >> 1;
                        // Update the neighboring label with the minimum value
                        n_label = (n_label == 0 || data < n_label) ? data : next_label;
                        // Update the equivalent list
                        write_map(n_label, data, 1, equ_list);
                        write_map(data, n_label, 1, equ_list);
                    }
                }

                // Check above pixel
                if (y > 0) {
                    index = DWNS_img_size * (y - 1) + x;
                    uint8_t data = input[index];
                    if (data & 0b00000001 == 1) {
                        data = data >> 1;
                        n_label = (n_label == 0 || data < n_label) ? data : n_label;
                        write_map(n_label, data, 1, equ_list);
                        write_map(data, n_label, 1, equ_list);
                    }
                }

                // Check above left pixel
                if (x > 0 && y > 0) {
                    index = DWNS_img_size * (y - 1) + x - 1;
                    uint8_t data = input[index];
                    if (data & 0b00000001 == 1) {
                        data = data >> 1;
                        n_label = (n_label == 0 || data < n_label) ? data : n_label;
                        write_map(n_label, data, 1, equ_list);
                        write_map(data, n_label, 1, equ_list);
                    }
                }

                // Check above right pixel
                if (x < DWNS_img_size - 1 && y > 0) {
                    index = DWNS_img_size * (y - 1) + x + 1;
                    uint8_t data = input[index];
                    if (data & 0b00000001 == 1) {
                        data = data >> 1;
                        n_label = (n_label == 0 || data < n_label) ? data : n_label;
                        write_map(n_label, data, 1, equ_list);
                        write_map(data, n_label, 1, equ_list);
                    }
                }

                // If no label found, assign a new one
                if (n_label == 0) {

```

```

        n_label = next_label++;
    }

    // Store the label in the reserved bits of the input pixel
    index = height * y + x;
    input[index] |= (n_label << 1);
    // Update the equivalent list for the current label
    if (n_label != 0) {
        write_map(n_label, n_label, 1, equ_list);
    }
}

// Second Pass: update equivalent list
// Loop through all the pixels in the input image
for (uint8_t y = 0; y < height; y++) {
    for (uint8_t x = 0; x < width; x++) {
        // Read the label value(bit 1-7) from the first pass
        uint8_t label = (input[height * y + x] & 0b1111110) >> 1;

        // Check if the label is not 0
        if (label != 0) {
            // Find the minimum index in the same label set based on equivalent list
            uint8_t min_l = label;
            for (uint8_t i = 1; i < cs_length; i++) {
                if (read_map(label, i, equ_list)) {
                    if (i < min_l) {
                        min_l = i;
                    }
                }
            }
            // If label is not the minimum label in the set, clear the label and update the input value
            if (label != min_l) {
                write_map(label, label, 0, equ_list);
                input[height * y + x] = (min_l << 1 | 0b00000001);
            }
        }
    }
}

uint8_t count = 0;
// Count the number of labels
for (uint8_t x = 0; x < cs_length; x++) {
    if (read_map(x, x, equ_list)) {
        count++;
    }
}

// Return the count of labels
return count;
}

```

Build Feature Set

```

//Loop through all possible labels and buil feature set
for(uint8_t c = 1; c < cs_length - 1; c++){// for all possible label
    // if the number of features exceed the limit of features , break out of the loop
    if(index > Max_features){
        break;
    }
    // initialize the sums for x coordinates , y coordinates , and area
    int sum_x = 0;
    int sum_y = 0;
    int sum = 0;
    // Loop through all pixels of the downsampling image
    for(int i = 0; i < DWNS_img_size; i++){
        for(int j = 0; j < DWNS_img_size; j++){
            // clear all buffer map pixels in previous loop
            write_map(j, DWNS_img_size - 1 - i, 0, buffer_map);
            // read 1-7 bits of the downsampling image as label number and check if they match the current label
            if((DWNS_img[i * DWNS_img_size + j] & 0b11111110) >> 1 == c){
                write_map(j, DWNS_img_size - 1 - i, 1, buffer_map); // segmentation , copy data into the buffer map
                sum++; // calculate area
                // accumulate the coordinate of segmented object and covert the coordinates fomr Raster to Cartesian
                sum_x += j;
                sum_y += (DWNS_img_size - 1 - i);
            }
        }
    }
    // if the area is large enough, consider it a valid feature
    if(sum >= 15){
        // get a pointer to the current feature and compute its descriptors
        feature* f = &feature_set[temp][index];
        f->descriptor[0] = perimeter(buffer_map, DWNS_img_size, DWNS_img_size);
        compute_shape_size(buffer_map, &f->descriptor[1], &f->descriptor[2]);
        f->descriptor[3] = sum;
        getHuMoments(buffer_map, DWNS_img_size, DWNS_img_size, &f->descriptor[4], &f->descriptor[5]);
        f->position_x = (int)(sum_x / sum);
        f->position_y = (int)(sum_y / sum);
        index++;
    }
}

```

ID-assignment

```

// Normalize feature descriptors using min-max scaling
for(uint8_t k = 0; k < Valid_descriptor; k++){ // for each descriptor
    double min_val = 65536;
    double max_val = 0;
    for(uint8_t i = 0; i < Max_maps; i++){ // for each map
        for(uint8_t j = 0; j < Max_features; j++){ // for each feature
            // find minimum and maximum of each descriptor of valid features(position isn't out of boundaries)
            if(feature_set[i][j].position_x != DWNS_img_size && feature_set[i][j].position_y != DWNS_img_size){
                double val = feature_set[i][j].descriptor[k];
                if(val < min_val){
                    min_val = val;
                }
                if(val > max_val){
                    max_val = val;
                }
            }
        }
    }
    // apply min-max scaling to each descriptor value for all features
    for(uint8_t i = 0; i < Max_maps; i++){ // for each map
        for(uint8_t j = 0; j < Max_features; j++){ // for each feature
            // normalize each descriptor of valid features(position isn't out of boundaries)
            if(feature_set[i][j].position_x != DWNS_img_size && feature_set[i][j].position_y != DWNS_img_size){
                double val = feature_set[i][j].descriptor[k];
                double normalized_val = (val - min_val) / (max_val - min_val);
                feature_set[i][j].descriptor[k] = normalized_val;
            }
        }
    }
}

// Assign feature ID based on similarity
uint8_t next_id = 0;
for(uint8_t i = 0; i < Max_maps; i++){ // for each map
    for(uint8_t j = 0; j < Max_features; j++){ // for each feature
        feature* f = &feature_set[i][j];
        // skip invalid features(position is not out of boundaries)
        if(f->position_x == DWNS_img_size || f->position_y == DWNS_img_size){
            continue;
        }
        double min_distance = 65536;
        int8_t nearest_id = -1;
        // compare each feature to all others and assign IDs based on similarity
        for(uint8_t k = 0; k < Max_maps; k++){ // for each map
            for(uint8_t l = 0; l < Max_features; l++){ // for each feature
                feature* other = &feature_set[k][l];
                if(other->position_x == DWNS_img_size || other->position_y == DWNS_img_size || other->id < 0){
                    continue;
                }
                double d = distance(f, other);
                if(d < min_distance){
                    min_distance = d;
                    nearest_id = other->id;
                }
            }
        }
        // assign ID to feature based on similarity
        if(min_distance < 0.7){
            f->id = nearest_id;
        }
        else{
            f->id = next_id++;
        }
    }
}

```

Calculation of the sum of the area and the Euclidean distance of each map and the angle difference

```

// Copy all features in small data set based on id and the position shifted to middle of image
for(uint8_t m = 0; m < 2; m++){ // loop through both feature sets
    for(uint8_t i = 0; i < Max_features; i++){ // loop through each feature in set
        feature* f = &feature_set[m][i]; // get a pointer to the feature
        // calculate x and y coordinates of feature
        Data_Points[m][0][i] = (f->position_x != DWNS_img_size) ? (f->position_x - (DWNS_img_size / 2)) : 0;
        Data_Points[m][1][i] = (f->position_y != DWNS_img_size) ? (f->position_y - (DWNS_img_size / 2)) : 0;
        // if feature has no ID, set z-coordinate to 0 and add distance to distance array
        if(f->id == -1){
            Data_Points[m][2][i] = 0;
            distance[m] += sqrt(f->position_x * f->position_x + f->position_y * f->position_y);
        }
        // if feature has an ID, set z-coordinate to ID and increment count
        else{
            Data_Points[m][2][i] = f->id;
            count[m] = count[m] + 1;
        }
        // add area to area array
        area[m] += f->descriptor[3];
    }
}
// Initialize variables
double min_angle = 0;
double min_distance_1 = 65536;
// Loop through different angles
for(double theta = 0; theta < 2 * 3.14; theta += 0.1){
    // Initialize total distance for this angle
    double total_distance = 0;

    // Loop through features in dataset 1
    for(int i = 0; i < Max_features; i++){
        // Extract coordinates for feature i
        double x = Data_Points[0][0][i];
        double y = Data_Points[0][1][i];
        double z = Data_Points[0][2][i];
        // Calculate sine and cosine of current angle
        double cos_theta = cos(theta);
        double sin_theta = sin(theta);
        // Apply rotation matrix to feature i
        double rx = cos_theta * x - sin_theta * y;
        double ry = sin_theta * x + cos_theta * y;
        double rz = z; // z coordinate is unchanged

        // Initialize minimum distance for this feature
        double min_distance_2 = 65536;
        // Loop through features in dataset 2
        for (int j = 0; j < Max_features; j++){
            // Check if feature j has the same z-coordinate as feature i
            if(z == Data_Points[1][2][j]){
                // Calculate distance between rotated feature i and feature j
                double dx = rx - Data_Points[1][0][j];
                double dy = ry - Data_Points[1][1][j];
                double d = sqrt(dx*dx + dy*dy);
                // Update minimum distance if necessary
                if(d < min_distance_2){
                    min_distance_2 = d;
                }
            }
        }
        // Add minimum distance for feature i to total distance
        total_distance = (min_distance_2 != 65536) ? (total_distance + min_distance_2) : total_distance;
    }

    // Update minimum distance and corresponding angle
    if (total_distance < min_distance_1){
        min_distance_1 = total_distance;
        min_angle = theta;
    }
}

```